

Docket No. AT 99-339

*Method and Apparatus in a Data Processing System for the Separation of
Role-Based Permissions Specification Form Its Corresponding Implementation
of Its Semantic Behavior*

~~MECHANISM FOR CROSS CHANNEL MULTI-SERVER MULTI-PROTOCOL
MULTI-DATA MODEL THIN CLIENTS~~

5

BACKGROUND OF THE INVENTION

1. Technical Field:

10 The present invention relates generally to an improved distributed data processing system and, in particular to an improved method and apparatus for creating applications. Still more particularly, the present invention relates to a method and apparatus for
15 creating client applications.

2. Description of Related Art:

20 Distributed data processing systems involve data transfers between clients and servers (also know as services). Typically, a client locates a server, initiates a session with a server and requests the server to perform some service. The server expects requests from a client to arrive in a particular format. A server is more complex than a client because the server
25 typically handles a large number of clients simultaneously, often fetches and stores information from a large database, creates additional transactions for other services, performs business logic, and returns information formatted according to each client channel.

30 For example, data will be specified in a particular message format. A particular transmission protocol will deliver the message to the server. The server accepts the message protocol as its application programming model (API) to its services and returns a result. A variety of
35 software systems, such as Enterprise Java Beans (EJB),

Servlets, Java Server Pages (JSP), and XML have been implemented to enhance the development of client and server-side software.

Client applications perform a number of different functions. For example, the application on the client side handles the user interface and may provide program logic for processing user input. Additionally, a client application must match the requirements of a particular server to provide communications with the particular server. Clients are packaged and distributed according to the services provided by the server.

A graphical user interface (GUI) exists in the client application to handle what the user views on the screen. Events resulting from user input, such as mouse clicks or keyboard strokes, are detected and handled using "listener" processes in the application. The events are processed by program logic. The program logic may result in requests being sent to a server. Communication with the server is provided using processes that use protocols, such as hypertext transfer protocol (HTTP), secure sockets (SSL), or Remote Method Invocation (RMI).

Client software can be either "thick" or "thin". A thick client is typically a large client-installed application that may access a database directly and apply business logic. They typically have dependence on the client operating system and require manual support to install and configure. By contrast a thin client is typically a small application downloaded on request from a server and accesses the database through an intermediate application server. This is known as a multi-tier application. A number of different usage

scenarios for clients are present, resulting in a variety of client needs being present. For example, it is typical that in an global enterprise Intranet, the client configuration is controlled by the business but the large number of clients includes older machines with slow networks (e.g. 9600 baud). Likewise, in the Internet, there is little configuration control by the business and it is estimated that a large percentage of clients worldwide still use 14.4K connections that result in very slow network speeds and downloads. A typical user will become very frustrated if downloads take longer than a minute or two. Further, mobile users require compact software that can be customized and packaged to fit on machines and operate disconnected from the network. Subsequent automated support to connect to the network is needed.

At the other end of the spectrum, power users with high speed connections expect screen refresh times in the sub-second range and "instantaneous" echoing of typed characters to provide the look and feel of processing in a local environment. In a multi-tier computing environment, the primary role of the client is to present and gather information quickly. The client application is considered a business asset independent of the network topology and server function. In these environments, it is desirable to be able to use the same client processing code for different user types and interface channels, such as automated teller machines (ATM), Kiosks, Internet [hypertext markup language (HTML)/ applets], and regional office clients (applications).

Consequently, a common thin or thick client development environment for developing clients may be

used to solve these problems, especially when the size and speed of the application download, integration and operation is important. Any software development environment should be based on sound software engineering principles.

Object-oriented languages have been employed in creating thin clients. Object-oriented programming environments have been presented as providing software reuse, which is a desirable feature in creating thin clients and reducing development time. In reality, the present object-oriented programming environments for developing thin clients are unable to provide enough object reuse and repeatability for quickly developing thin clients. Nor do they specify how to readily support additional message formats, protocols, data models and servers, mobile disconnected users, and caching.

Therefore, it would be advantageous to have an improved method and apparatus for a client development architecture that facilitates creating thin clients in a manner in which component reuse is increased while client development time is reduced, and multiple message formats, protocols, data models and servers, mobile disconnected users and caching can be readily integrated.

SUMMARY OF THE INVENTION

The present invention provides an architectural pattern for creating applications for a data processing system. A graphical user interface is created in which the graphical user interface includes a plurality of components. Processes for presenting the plurality of components and receiving user input are handled by a first set of graphical objects, wherein in response to selected user input, a first event is generated. An application object is created in which the application process controls an order in which the graphical objects present the set of components and process the event and wherein the application generates a second event. A transport object is created in which the transport object processes the second event and forwards the second event for processing to a destination within the plurality of destinations. A plurality of destination objects are created in which each destination object within the plurality of destinations objects handles accessing a destination within the plurality of destinations.

The present invention provides a method and apparatus in a data processing system for refreshing data in an application. A call is received to update data in the application, wherein the data is destined for a component in the application. A data type is identified for the data. Responsive to the data type being a handled data type, the data is formatted and a refresh is called on the component.

The present invention provides a method and apparatus in a data processing system for displaying a component or container. The container is displayed within a display

using a first component. A location of the component or container is controlled within the display using a second component, wherein the second component controls the location and geometry of the component or container in response to receiving an event. The component or container is selectively displayed using a third component, wherein the third component generates the event.

The present invention provides a process in a data processing system for managing services in a desktop environment from an object oriented-environment. A presentation of a graphical user interface is controlled using a view controller, wherein the view controller handles user input to the graphical user interface. Responsive to a selected user input, the selected user input is sent from the view controller to an application mediator. Responsive to receiving the selected user input at the application mediator, the selected user input is processed at the application mediator. Responsive to the application mediator determining that a service is required in the desktop environment, an event is generated. Responsive to detecting the event at a listener object, a method is executed in the listener object to perform the service in the desktop environment.

The present invention provides a method and apparatus in a data processing system for managing transactions. A request event is received at a transporter object. The request event includes a target and an indication of how to handle the request event. A destination object is identified within the plurality of destination objects using the request event to form an identified destination object. The request event is sent to the identified destination object, wherein the

identified destination object handles the request using the indication and accesses the target.

5 The present invention provides a method and apparatus in a data processing system for displaying a graphical user interface. A container is displayed in a graphical user interface from a set of containers, wherein a display of the container handled by a view controller from a set of view controllers. Each view controller handles the display of an associated container within the set of containers and user input for the associated container. A display of the set of containers is altered by an application mediator, wherein the set of containers are displayed in an order determined by the application mediator.

15 The present invention provides a method and apparatus in a data processing system for performing validation of user input. User input is received in a container displayed in a graphical user interface, wherein presentation of the container and the user input to the container are handled by a view controller. Responsive to receiving the user input, a call is sent to a validation object by the view controller. Responsive to the call, the validation object tests the user input using a criteria, wherein the rule is separate from the view controller.

25 The present invention provides a method and apparatus in a data processing system for managing permissions in an application. A user input is received at a container handled by a view controller, wherein the user input requests a change in permissions in the application. This user input, may be, for example, a change in security in an application through a login process. A view event describing the user input is

30

generated. The view event is received at an application mediator. Responsive to receiving the view event, by the application mediator, a request event is generated and a permission corresponding to the user input is received.

- 5 The permission alters an item, which may be in either of both the view controller and the application mediator.

- The present invention provides a process and apparatus in a data processing system for presenting a view to a client. At an application mediator, a view
- 10 event is received from a view controller, wherein the view event describes an action on a displayed container handled by the view controller. Responsive to a requirement that a change in a placement of the displayed container is required, a placement event is generated by
- 15 the application mediator. A determination is then made by a placement listener, as to whether the placement event includes an indication that an alternate view is to be generated. Responsive to a determination that an alternate view is to be generated, a call is sent to a
- 20 method in the view controller to generate the alternate view.

- The present invention provides a method and apparatus in a data processing system for processing user input in a graphical user interface. A graphical user
- 25 interface is presented using a view controller, wherein the view controller handles the user input to the graphical user interface. Responsive to a selected user input, an event is sent to a first application mediator. Responsive to the first application mediator being unable
- 30 to process the event, the event is sent to a second application mediator for processing, wherein the first application mediator and the second application mediator

handle an order in which a set of displays are displayed by a view controller.

The present invention provides a method and apparatus in a data processing system for presenting a set of screens in a graphical user interface. A first screen within a set of screens is presented, wherein the set of screens are presented using a set of view controllers. Responsive to a selected user input to the first screen, an event is generated by a view controller within the set of view controllers identifying the user input to the first screen, which is handled by the first view controller. Responsive to detecting the event generated by the view controller, a second screen from the set of screens is selected, by an application mediator, for display by sending a response to a view controller handling the second screen.

The application mediator is initialized from reading a state machine file and control processing of view event received from virtual controllers.

The present invention provides a method and apparatus in a data processing system for serializing data. A serializer receives a data element for serialization, wherein the data element includes a class name string. Responsive to receiving the data element, the serializer replaces the class name string with a code having a smaller size than the class name string to form a modified data element. Responsive to forming the modified data element, in which the serializer serializes the modified data element. This serialized data is transmitted and deserialized by a deserializer, which replaces the indicator with the class name.

The present invention provides a method and

apparatus in a data processing system for providing an interface to an application for monitoring execution of the application. An event generated by a view controller is detected, wherein the view controller handles
5 presentation of a container in a graphical user interface. A determination is made as to whether the event is an event selected for monitoring. Responsive to the determination that the event is an event selected for monitoring, a request event is generated, wherein the
10 request event includes data from the event and a destination.

The present invention provides a method and apparatus for a data processing system for accessing classes and methods in an object oriented system.
15 Responsive to receiving a selected user input to a container, a view event is sent from a view controller to an application mediator. The view event identifies an action taken to generate the selected user input. A request is selectively generated based on the view event,
20 wherein the request event includes a major code identifying a class name as a destination and a minor code identifying a method name a function to be invoked. The request event is sent to a transporter. The transporter acts as a router to send the request event to
25 an appropriate destination object from a plurality of destination objects. Responsive to receiving the request event at the transporter, the request event is sent to a destination object within a plurality of destination objects based in the class name. The destination object
30 formats the request event into a form recognizable by the destination associated with the destination object. The destination may be located on a remote data processing

system. The request event is used to access the class or method identified in the request event. The access may be, for example, an invocation of the method.

BRIEF DESCRIPTION OF THE DRAWINGS

The novel features believed characteristic of the invention are set forth in the appended claims. The invention itself, however, as well as a preferred mode of use, further objectives and advantages thereof, will best be understood by reference to the following detailed description of an illustrative embodiment when read in conjunction with the accompanying drawings, wherein:

10. **Figure 1** depicts a pictorial representation of a distributed data processing system in which the present invention may be implemented;

Figure 2 is a block diagram depicting a data processing system that may be implemented as a server in accordance with a preferred embodiment of the present invention;

Figure 3 is a block diagram illustrating a data processing system in which the present invention may be implemented;

20. **Figure 4** is a diagram illustrating a model view controller diagram depicted in accordance with a preferred embodiment of the present invention;

Figure 5 is a diagram illustrating the components in an architectural pattern depicted in accordance with a preferred embodiment of the present invention;

Figure 6 is a diagram illustrating classes in a class hierarchy depicted in accordance with a preferred embodiment of the present invention;

30. **Figure 7** is a unified modeling language diagram depicted in accordance with a preferred embodiment of the present invention;

Figures 8A and 8B are diagrams illustrating

variables and methods in a ViewController depicted in accordance with a preferred embodiment of the present invention;

5 **Figures 9A-9C** are diagrams illustrating variables, constructors, and methods for ViewControllerImpl depicted in accordance with a preferred embodiment of the present invention;

10 **Figures 10A-10C** are tables which show the variables, constructors and methods for ViewControllerBaseImpl depicted in accordance with a preferred embodiment of the present invention;

15 **Figures 11A-11C** are tables which illustrate a variable, a constructor, and methods for a ViewControllerAdapter depicted in accordance with a preferred embodiment of the present invention;

Figures 12A-12D are drawings illustrating variables, constructors, and methods for a ValidationRule depicted in accordance with a preferred embodiment of the present invention;

20 **Figures 13A and 13B** are tables illustrating variables and constructors for a ValidationRuleException depicted in accordance with a preferred embodiment of the present invention;

25 **Figures 14A-14F** are diagrams illustrating variables, constructors, and methods for a ViewEvent depicted in accordance with a preferred embodiment of the present invention;

30 **Figures 15A and 15B** are diagrams illustrating a variable and a method for a ViewListener depicted in accordance with a preferred embodiment of the present invention;

Figures 16A and 16B diagrams illustrating a variable

and methods for an ApplicationMediator depicted in accordance with a preferred embodiment of the present invention;

5 Figure 17A-17D are diagrams illustrating variables and a constructor for an ApplicationMediatorImpl depicted in accordance with a preferred embodiment of the present invention;

10 Figures 17E-17H are diagrams illustrating code used in methods for an ApplicationMediatorImpl depicted in accordance with a preferred embodiment of the present invention;

 Figures 17I-17L are diagrams which depicts code used to handle the event dispatch in accordance with a preferred embodiment of the present invention;

15 Figures 18A-18C are diagrams illustrating variables, constructors, and methods for a PlacementEvent depicted in accordance with a preferred embodiment of the present invention;

20 Figures 19A and 19B are diagrams illustrating a variable and method for a PlacementListener depicted in accordance with a preferred embodiment of the present invention;

25 Figures 20A-20C are diagrams illustrating variables, constructors, and methods for a TopEvent depicted in accordance with a preferred embodiment of the present invention;

30 Figure 21A and 21B are diagrams illustrating a variable and methods for TopListeners depicted in accordance with a preferred embodiment of the present invention;

 Figures 22A-22C are diagrams illustrating a variable, constructors, and methods for RequestEvent

depicted in accordance with a preferred embodiment of the present invention;

Figures 23A-23C are diagrams illustrating a variable, constructors, and methods for a
5 RequestException depicted in accordance with a preferred embodiment of the present invention;

Figures 24A and 24B are diagrams illustrating a variable and methods for a RequestListener depicted in accordance with a preferred embodiment of the present
10 invention;

Figures 25A and 25B are diagrams illustrating a variable and methods for a RequestResponseListener depicted in accordance with a preferred embodiment of the present invention;

Figures 26A-26C are diagrams illustrating variables, a constructor, and methods for a Transporter depicted in accordance with a preferred embodiment of the present
15 invention;

Figures 26D-26G are diagrams illustrating code used in methods in a Transporter depicted in accordance with a preferred embodiment of the present invention;
20

Figures 27A and 27B are diagrams illustrating a variable and methods for a Destination depicted in accordance with a preferred embodiment of the present
25 invention;

Figures 28A-28C are diagrams illustrating variables, constructors, and methods for a DestinationImpl depicted in accordance with a preferred embodiment of the present invention;

Figure 28D is code used to process a RequestEvent depicted in accordance with a preferred embodiment of the present invention;
30

Figures 29A and 29B are diagrams illustrating variables and methods in a Factory depicted in accordance with a preferred embodiment of the present invention;

Figures 30A and 30B are diagrams illustrating a
5 variable and methods for a JTC depicted in accordance with a preferred embodiment of the present invention;

Figure 31 is a flowchart of a process for creating a ViewController depicted in accordance with a preferred embodiment of the present invention;

10 Figure 32 is a flowchart of a process for creating ValidationRules depicted in accordance with a preferred embodiment of the present invention;

Figure 33 is a flowchart of a process for creating a ViewEvent depicted in accordance with a preferred
15 embodiment of the present invention;

Figure 34 is a flowchart of a process to create an ApplicationMediator depicted in accordance with a preferred embodiment of the present invention;

Figure 35 is a flowchart of a process for creating a
20 RequestEvent depicted in accordance with a preferred embodiment of the present invention;

Figure 36 is a flowchart of a process for creating a Destination depicted in accordance with a preferred embodiment of the present invention;

25 Figure 37 is a flowchart of a process for creating a TopListener depicted in accordance with a preferred embodiment of the present invention;

Figure 38 is a flowchart of a process for creating a PlacementListener depicted in accordance with a preferred
30 embodiment of the present invention;

Figure 39 is a diagram illustrating runtime behavior

of a ViewController subsystem depicted in accordance with a preferred embodiment of the present invention;

Figure 40 are steps in the operation of a ViewController subsystem, as viewed from a ViewControllerImpl, depicted in accordance with a preferred embodiment of the present invention;

Figure 41 is a flowchart of a process for a JTC application to present alternate views (HTML/XML) of itself while running in a separate environment, such as a server, as the alternate views depicted in accordance with a preferred embodiment of the present invention;

Figures 42 and 43 are diagrams detailing processes within the ViewController subsystem depicted in accordance with a preferred embodiment of the present invention;

Figure 44 is a complete list of predefined major event codes depicted in accordance with a preferred embodiment of the present invention;

Figure 44 shows how a text field representing a social security number can be validated and displayed depicted in accordance with a preferred embodiment of the present invention;

Figure 46 shows how a social security number can be validated and converted back to a transmittable format depicted in accordance with a preferred embodiment of the present invention;

Figure 47 illustrates the application of two edit rules, range checking, and formatting for viewing depicted in accordance with a preferred embodiment of the present invention;

Figure 48 illustrates inheritance where the

ViewControllerBaseImpl is a subclass of JPanel from the Java swing components depicted in accordance of a preferred embodiment of the present invention;

Figure 49 illustrates inheritance where the
5 ViewControllerBaseImpl is a subclass of java.lang.Object and where the methods getComponent, setEnabled and setVisible are implemented to ensure the ViewController subclassing ViewControllerBaseImpl can be treated as a GUI component, container or bean depicted in accordance
10 of a preferred embodiment of the present invention;

Figure 50 is a diagram illustrating runtime behavior of an ApplicationMediator subsystem depicted in accordance with a preferred embodiment of the present invention;

15 Figure 51 is a diagram illustrating Event threading support depicted in accordance with a preferred embodiment of the present invention;

Figure 52 is a flowchart of a process used in designing and executing an ApplicationMediator depicted in
20 accordance with a preferred embodiment of the present invention;

Figure 53 is a diagram illustrating runtime behavior of the Placement subsystem is depicted in accordance with a preferred embodiment of the present invention;

25 Figure 54 is Java code illustrating the creation and firing of a PlacementEvent in an ApplicationMediator depicted in accordance with a preferred embodiment of the present invention;

Figure 55 is Java code illustrating the callback to
30 a PlacementListener with a PlacementEvent and inspecting the PlacementEvent for semantic interpretation depicted

in accordance with a preferred embodiment of the present invention;

Figure 56 is a flowchart of a process used in processing a PlacementEvent depicted in accordance with a preferred embodiment of the present invention;

Figure 57 is a diagram illustrating runtime behavior for a TopListener subsystem depicted in accordance with a preferred embodiment of the present invention;

Figure 58 is Java code illustrating the creation of an ApplicationMediator and the adding of a PlacementListener depicted in accordance with a preferred embodiment of the present invention;

Figure 59, is Java code illustrating the creation and firing of a TopEvent in an ApplicationMediator depicted in accordance with a preferred embodiment of the present invention;

Figure 60 is Java code illustrating the callback to a TopListener with a TopEvent and inspecting the TopEvent for semantic interpretation depicted in accordance with a preferred embodiment of the present invention;

Figure 61 is a diagram illustrating runtime behavior of a RequestEvent subsystem depicted in accordance with a preferred embodiment of the present invention;

Figure 62 is Java code illustrating the creation of RequestEvent, setting its major and minor codes, and firing an asynchronous RequestEvent from an ApplicationMediator depicted in accordance with a preferred embodiment of the present invention;

Figure 63 is Java code illustrating the callback, to an ApplicationMediator, a successful asynchronous RequestEvent with a result depicted in accordance with a

preferred embodiment of the present invention;

Figure 64 is Java code illustrating the callback, to an ApplicationMediator, an unsuccessful asynchronous RequestEvent with a RequestException depicted in accordance with a preferred embodiment of the present invention;

Figure 65 is a flowchart of a process for using a RequestEvent depicted in accordance with a preferred embodiment of the present invention;

10 Figure 66 is a diagram illustrating runtime behavior of a Transporter subsystem depicted in accordance with a preferred embodiment of the present invention;

Figure 67 is Java code illustrating creation of a Transporter and adding it as a RequestListener depicted in accordance with a preferred embodiment of the present invention;

Figure 68 is runtime behavior of a Destination subsystem shown in accordance with a preferred embodiment of the present invention;

20 Figure 69 is a diagram of Java code for creation of a Destination, setting a major code and adding it to a Transporter as a DestinationListener depicted in accordance with a preferred embodiment of the present invention;

25 Figure 70 is a diagram illustrating Java code for creating Destinations with wild card, priority and normal major codes, firing a RequestEvent, and a report of the expected results depicted in accordance with a preferred embodiment of the present invention;

30 Figure 71 is a diagram of Java code used for accessing, identifying type and recursively attaching JTC,

AWT and JFC listeners to JTC, AWT and JFC programs and objects depicted in accordance with a preferred embodiment of the present invention;

5 Figure 72 is a diagram of Java code used for attaching JTC listeners to JTC ApplicationMediators, ViewControllers and Transporters depicted in accordance with a preferred embodiment of the present invention;

10 Figure 73 is a diagram of Java code used for attaching AWT and JFC listeners to AWT and JFC containers, components and beans depicted in accordance with a preferred embodiment of the present invention;

15 Figure 74 is a diagram of Java code used for attaching AWT and JFC listeners to AWT and JFC components (java.awt.Button, com.sun.swing.java.JButton and com. and com.sun.java.swing.JTextField) depicted in accordance with a preferred embodiment of the present invention;

20 Figure 75 is a flowchart of a process for performing hookJTCs and hookAWTs, non intrusive tracing, hooking, debugging, monitoring and logging of JTC programs and JTC program objects depicted in accordance with a preferred embodiment of the present invention;

Figure 76 is a diagram describing the relationship between components and containers depicted in accordance with a preferred embodiment of the present invention;

25 Figure 77 is a flowchart of a process for performing hookAWTs depicted in accordance with the preferred embodiment of the present invention;

30 Figure 78 is a process for hooking an AWT and JFC component depicted in accordance with the preferred embodiment of the present invention;

Figures 79-82 are diagrams of Java code for use in

managing the refresh of data objects in an
ApplicationMediator (79), managing the refresh of data
objects in a ViewController using initial types (80), and
managing the refresh of data objects in a ViewController
5 using an multiple concurrent updated or additional data
types (81, 82) depicted in accordance with a preferred
embodiment of the present invention;

Figure 83 is a diagram of using multiple concurrent
data model update notification mechanisms in a
10 ViewController and ApplicationMediator depicted in
accordance with a preferred embodiment of the present
invention;

Figure 84 is a flowchart of a process used in a
TopListener depicted in accordance with a preferred
15 embodiment of the present invention;

Figure 85 is a flowchart of a PlacementListener
depicted in accordance with a preferred embodiment of the
present invention;

Figure 86 is a flowchart illustrating handling of
20 AWTEvents by a ViewController depicted in accordance with
a preferred embodiment of the present invention;

Figure 87 is a flowchart illustrating application of
ValidationRules depicted in accordance with a preferred
embodiment of the present invention;

25 Figure 88 is a flowchart of a process for firing a
ViewEvent depicted in accordance with a preferred
embodiment of the present invention;

Figures 89-95 are flowcharts illustrating processes
used by an ApplicationMediator depicted in accordance
30 with a preferred embodiment of the present invention;

Figures 96 and 97 are diagrams illustrating

processes used to refresh object data in an ApplicationMediator and ViewController depicted in accordance with a preferred embodiment of the present invention;

- 5 Figure 98 is a flowchart of a process used to process RequestEvents depicted in accordance with a preferred embodiment of the present invention;

- 10 Figure 99 is a flowchart of an initialization process for creating hierarchical ApplicationMediators depicted in accordance with a preferred embodiment of the present invention;

- 15 Figure 100 is a flowchart of a process for handling events in a hierarchical ApplicationMediator system depicted in accordance with a preferred embodiment of the present invention;

Figure 101 is a flowchart for a process for building a virtual ApplicationMediator state dispatching machine depicted in accordance with a preferred embodiment of the present invention;

- 20 Figure 102 illustrates example table entries from the loading of a configuration file for a virtual ApplicationMediator state machine depicted in accordance of a preferred embodiment of the present invention;

- 25 Figures 103 and 104 are virtual ApplicationMediator access state dispatching machine used to determine whether processing of a ViewEvent is needed depicted in accordance with a preferred embodiment of the present invention;

- 30 Figure 105 is a diagram of a serializer system write depicted in accordance with a preferred embodiment of the present invention;

Figure 106 is a diagram of a serializer system read depicted in accordance with a preferred embodiment of the present invention;

Figure 107 is a diagram illustrating an object array depicted in accordance with a preferred embodiment of the present invention;

Figure 108 is a diagram illustrating code used in a serialization method depicted in accordance with a preferred embodiment of the present invention;

10 Figure 109 implements the methods readExternal and writeExternal for reading/writing from/to input/output stream depicted in accordance of a preferred embodiment of the present invention;

Figure 110 is a diagram illustrating code used in a serialization method depicted in accordance with a preferred embodiment of the present invention;

Figure 111 is a flowchart of a process for using a serializer depicted in accordance with the preferred embodiment of the present invention;

20 Figure 112 is a flowchart of a process for statically creating matching user profiles and associated JTC permission keys for JTC programs, where keys are built by recursively querying JTC ApplicationMediator and ViewController objects, depicted in accordance with a preferred embodiment of the present invention;

Figure 113 is a flowchart of a process for a JTC program building a database of permission keys by iterating over JTC ApplicationMediator's getPermissions method and returning permission keys that are
30 ViewController names that are alterable at runtime depicted in accordance with a preferred embodiment of the

present invention;

Figure 114 is a flowchart of a process for a JTC ApplicationMediator building a database of permission keys by iterating over JTC ViewController's

- 5 getPermissions method and returning permission keys that are names known only to the ViewController and that are alterable at runtime depicted in accordance with a preferred embodiment of the present invention;

- Figure 115 is a flowchart of a process for a JTC
10 ViewController building a database of permission keys by iterating over runtime alterable components, containers and beans and returning permission keys that are component, container or bean names that are alterable at runtime depicted in accordance with a preferred
15 embodiment of the present invention;

- Figures 116 is a flowchart of a process for a JTC program accepting a database of permission keys by retrieving permission key/value data supplying the data to a JTC program depicted in accordance with a preferred
20 embodiment of the present invention;

- Figure 117 is a flowchart of a process for a JTC program supplied with a setting of permission key/value data and iterating over all ApplicationMediators and passing the key/value data through setPermissions
25 depicted in accordance with a preferred embodiment of the present invention;

- Figure 118 is a flowchart of a process for a JTC ApplicationMediator called with setPermissions of permission key/value data and iterating over all
30 ViewControllers and passing the key/value data through setPermissions depicted in accordance with a preferred embodiment of the present invention;

Figure 119 is a flowchart of a process for a JTC ViewController called with setPermissions of permission key/value data and iterating over the permission keys, identifying the corresponding components, containers and beans, and applying the value to the components, containers and beans, depicted in accordance with a preferred embodiment of the present invention; and

Figures 120-123 illustrate example patterns using the architectural pattern of the present invention depicted in accordance of a preferred embodiment of the present invention.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENT

I. Hardware

With reference now to the figures, Figure 1 depicts a pictorial representation of a distributed data processing system in which the present invention may be implemented. Distributed data processing system 100 is a network of computers in which the present invention may be implemented. Distributed data processing system 100 contains a network 102, which is the medium used to provide communications links between various devices and computers connected together within distributed data processing system 100. Network 102 may include permanent connections, such as wire or fiber optic cables, or temporary connections made through telephone connections.

In the depicted example, a server 104 is connected to network 102 along with storage unit 106. In addition, clients 108, 110, and 112 also are connected to a network 102. These clients 108, 110, and 112 may be, for example, personal computers or network computers. For purposes of this application, a network computer is any computer, coupled to a network, which receives a program or other application from another computer coupled to the network. In the depicted example, server 104 provides data, such as boot files, operating system images, and applications to clients 108-112. Clients 108, 110, and 112 are clients to server 104. Distributed data processing system 100 may include additional servers, clients, and other devices not shown. In the depicted example, distributed data processing system 100 is the Internet with network 102 representing a worldwide collection of networks and

gateways that use the TCP/IP suite of protocols to communicate with one another. At the heart of the Internet is a backbone of high-speed data communication lines between major nodes or host computers, consisting of
5 thousands of commercial, government, educational and other computer systems that route data and messages. Of course, distributed data processing system 100 also may be implemented as a number of different types of networks, such as for example, an intranet, a local area network
10 (LAN), or a wide area network (WAN). Figure 1 is intended as an example, and not as an architectural limitation for the present invention.

Referring to Figure 2, a block diagram depicts a data processing system that may be implemented as a server,
15 such as server 104 in Figure 1, in accordance with a preferred embodiment of the present invention. Data processing system 200 may be a symmetric multiprocessor (SMP) system including a plurality of processors 202 and 204 connected to system bus 206. Alternatively, a single
20 processor system may be employed. Also connected to system bus 206 is memory controller/cache 208, which provides an interface to local memory 209. I/O bus bridge 210 is connected to system bus 206 and provides an interface to I/O bus 212. Memory controller/cache 208 and
25 I/O bus bridge 210 may be integrated as depicted.

Peripheral component interconnect (PCI) bus bridge 214 connected to I/O bus 212 provides an interface to PCI local bus 216. A number of modems may be connected to PCI bus 216. Typical PCI bus implementations will support
30 four PCI expansion slots or add-in connectors. Communications links to network computers 108-112 in

Figure 1 may be provided through modem 218 and network adapter 220 connected to PCI local bus 216 through add-in boards.

Additional PCI bus bridges 222 and 224 provide
5 interfaces for additional PCI buses 226 and 228, from which additional modems or network adapters may be supported. In this manner, server 200 allows connections to multiple network computers. A memory-mapped graphics adapter 230 and hard disk 232 may also be connected to I/O
10 bus 212 as depicted, either directly or indirectly.

Those of ordinary skill in the art will appreciate that the hardware depicted in Figure 2 may vary. For example, other peripheral devices, such as optical disk drives and the like, also may be used in addition to or in
15 place of the hardware depicted. The depicted example is not meant to imply architectural limitations with respect to the present invention.

The data processing system depicted in Figure 2 may be, for example, an IBM RISC/System 6000 system, a product
20 of International Business Machines Corporation in Armonk, New York, running the Advanced Interactive Executive (AIX) operating system.

With reference now to Figure 3, a block diagram illustrates a data processing system in which the present
25 invention may be implemented. Data processing system 300 is an example of a client computer. Data processing system 300 employs a peripheral component interconnect (PCI) local bus architecture. Although the depicted example employs a PCI bus, other bus architectures such as
30 Micro Channel and Industry Standard Architecture (ISA) may be used. Processor 302 and main memory 304 are connected

to PCI local bus 306 through PCI bridge 308. PCI bridge 308 also may include an integrated memory controller and cache memory for processor 302. Additional connections to PCI local bus 306 may be made through direct component interconnection or through add-in boards. In the depicted example, local area network (LAN) adapter 310, Small Computer System Interface host bus adapter 312, and expansion bus interface 314 are connected to PCI local bus 306 by direct component connection. In contrast, audio adapter 316, graphics adapter 318, and audio/video adapter 319 are connected to PCI local bus 306 by add-in boards inserted into expansion slots. Expansion bus interface 314 provides a connection for a keyboard and mouse adapter 320, modem 322, and additional memory 324. Small computer system interface (SCSI) host bus adapter 312 provides a connection for hard disk drive 326, tape drive 328, and CD-ROM drive 330. Typical PCI local bus implementations will support three or four PCI expansion slots or add-in connectors.

An operating system runs on processor 302 and is used to coordinate and provide control of various components within data processing system 300 in Figure 3. The operating system may be a commercially available operating system such as OS/2, which is available from International Business Machines Corporation. "OS/2" is a trademark of International Business Machines Corporation. An object oriented programming system such as Java may run in conjunction with the operating system and provides calls to the operating system from Java programs or applications executing on data processing system 300. "Java" is a trademark of Sun Microsystems, Inc. Instructions for the

operating system, the object-oriented operating system, and applications or programs are located on storage devices, such as hard disk drive 326, and may be loaded into main memory 304 for execution by processor 302.

5 Those of ordinary skill in the art will appreciate that the hardware in **Figure 3** may vary depending on the implementation. Other internal hardware or peripheral devices, such as flash ROM (or equivalent nonvolatile memory) or optical disk drives and the like, may be used
10 in addition to or in place of the hardware depicted in **Figure 3**. Also, the processes of the present invention may be applied to a multiprocessor data processing system.

For example, data processing system 300, if
15 optionally configured as a network computer, may not include SCSI host bus adapter 312, hard disk drive 326, tape drive 328, and CD-ROM 330, as noted by dotted line 332 in **Figure 3** denoting optional inclusion. In that case, the computer, to be properly called a client
20 computer, must include some type of network communication interface, such as LAN adapter 310, modem 322, or the like. As another example, data processing system 300 may be a stand-alone system configured to be bootable without relying on some type of network communication interface,
25 whether or not data processing system 300 comprises some type of network communication interface. As a further example, data processing system 300 may be a Personal Digital Assistant (PDA) device which is configured with ROM and/or flash ROM in order to provide nonvolatile
30 memory for storing operating system files and/or user-generated data.

The depicted example in **Figure 3** and above-described

examples are not meant to imply architectural limitations. For example, data processing system 300 also may be a notebook computer or hand held computer in addition to taking the form of a PDA. Data processing
5 system 300 also may be a kiosk or a Web appliance.

II. Overview

With reference next to Figure 4, a diagram illustrating a model view controller diagram is depicted
10 in accordance with a preferred embodiment of the present invention. Model view control (MVC) diagram 400 includes an end-to-end section 402 in which the view is represented by client 404, the control is represented by mid-tier logic 406 and the model is represented by
15 persistence storage 408. Section 410 is a graphical user interface (GUI) component architecture in which the view are Java components 412, the control are event listeners 414, and the model are Java models 416. GUI components 410 may be currently embodied in systems such as the Java
20 Foundation Classes (JFC) from Java.

In accordance with a preferred embodiment of the present invention, the present invention provides an architectural pattern for views in the client, for navigation of views in the client, for placing and
25 presenting views in a client, for issuing requests for different concurrent servers and services from the client, for issuing requests for client platform services from the client, for using multiple concurrent data model types in the client, for issuing multiple message
30 formats from the client, for using multiple protocols in the client and for specific partitioning of these pattern

components in the client, such as client 404. In particular, section 418 in these examples significantly defines and enhances and includes a view as Java Abstract Windows Toolkit (AWT)/(JFC) 420, a control as screen control 422, and a model as transactions 424. AWT is a tool kit containing primitives for basic windowing functionality. These primitives include user interface functionality, such as window and dialogue box manipulations, text rendering, buttons, check box, and radio button creation and manipulation, as well as graphics primitives such as line drawing, color choice. Virtually all sophisticated graphics and user-interface tools are built upon these primitives. The Java foundation classes (JFC) is a package containing, among other things, primitives for windowing functionality that provide a rich superset of AWT. These primitives or components include everything that the AWT provides along with a rich superset of new primitives, including, for example, tree view, tabbed panes, hypertext markup language (HTML) text views, etc. Java AWT/(JFC) 420 could be described through section 410.

Object-oriented design depends heavily on the class concept and the relationships between various classes. A class is an abstraction of an object that contains both attributes (data) and behavior (methods). Each new object created from a class is referred to as an instance of that class. In other words, the class is an abstract encapsulation mechanism while an instance is a particular object created from the class. If a method in a class is called, it is sometimes said a message is sent to an object.

Classes are organized into inheritance hierarchies

where a parent class may have one or several subclasses. The subclasses share all the data and methods from the parent class and any other ancestor class in the inheritance tree. The search for the appropriate methods to be used starts at the class for the object and proceeds up the tree as necessary to find the desired method (the name and parameters of the method must match exactly). Multiple inheritance means that a class can have two or more parent classes. Some programming languages (e.g., C++) support multiple inheritance while others (e.g., Smalltalk) do not. Multiple inheritance is difficult to manage, so some languages (e.g., Java) provide a mechanism called interfaces that provide a type oriented mechanism to achieve a similar functionality without incurring the implementation burdens. The class hierarchy diagram is based on a Java implementation and makes extensive use of the interface mechanism.

III. Architectural Pattern

The present invention provides a method, apparatus, and instructions for an architectural programming pattern and implementation. The architectural pattern of the present invention is illustrated as a Java implementation for building thin (or thick) client applications and is also referred to as "JTC." JTC is a process, architectural pattern, and implementation to guide developers on how to build applications, and in particular, Internet style thin clients. JTC adapts to Internet changes rapidly. JTC increases developer discipline by providing a common repeatable programming pattern. JTC facilitates project management by providing for concurrent development of the client. JTC

facilitates project management by providing for concurrent client and server(s) development. JTC provides for a natural partition of various programmer skills. JTC has a formula based approach for cost estimates. JTC provides multi-channel support (ATM, Internet, Kiosk, Extranet, Mobile, Branch, Call Center, Business partners, etc.). JTC supports mobile user disconnected mode. JTC provides natural support for multiple servers with reduced code level cohesion and coupling. JTC provides natural support for multiple data models such as, for example, XML, DHTML, Objects, Key/Value, EJB, streaming and asynchronous. JTC has natural support for multiple protocols, such as, for example IIOP, RMI, Sockets, HTTP, HTTPs, and Files. JTC has natural support for client Java accessible RS232 devices and APIs such as currency counters, J/XFS, printers, and coin dispensers.

With reference now to Figure 5, a diagram illustrating the components in an architectural pattern is depicted in accordance with a preferred embodiment of the present invention. Architectural pattern 500 includes a ViewController 502, which provides a display of a component, container or bean on a data processing system. ViewController 502 basically provides a reusable GUI element containing graphical components such as text fields, labels, buttons, tables, images, beans, etc. to be displayed for viewing and interacting by a user. ValidationRules 504 are used to validate and format the user entry into components contained within the ViewController. For example, ValidationRules 504 may be applied to a single text field or to groups of text fields used for user entry. If a violation of a rule

occurs, ValidationRuleException 506 is generated.

User inputs occur on the components, containers and beans controlled by ViewController 502. This user input is in the form of an AWTEvent 508 in the depicted examples. The AWTEvent 508 may be, for example, the click of a mouse button selecting a graphical button such as an "okay" button. In response, ViewController 502 will fire ViewEvent 510. ViewEvent 510 will contain major and minor codes that add more semantics than the AWTEvent 508. The changed data that may have been modified or entered by user or a reference to the data model may also be supplied for further semantics. AWTEvents 508 are not visible outside of the ViewController 502.

Next in architectural pattern 500, ApplicationMediator 512 is a form of a ViewListener in the depicted examples. ApplicationMediator 512 is the interface used to specify default behavior to control ViewController 502 as well as other ApplicationMediators 512. ApplicationMediator 512 will also mediate between ViewController 502 generated ViewEvents 510 and PlacementListener 514, TopListener 516 and Transporter 524. In this pattern, PlacementListener 514 is employed to manage the placement/containment of ViewControllers, such as ViewController 512 on the screen of a computer. PlacementListener 514 will manage ViewController 502 in response to a PlacementEvent 518 received from ApplicationMediator 512. For example, for a given component, container or bean represented by ViewController 502, PlacementListener 514 will control how ViewController 502 is placed on a screen, such as

possibly in the south of a frame with a border layout.

ApplicationMediator 512 controls the ordering of ViewControllers, not the placement of ViewControllers. As a result, complete separation between view creation and layout by ViewController 502, view ordering and mediation by ApplicationMediator 512, and view placement by PlacementListener 514 is provided. This mechanism in architectural pattern 500 increases the reusability of ViewControllers, ApplicationMediators, and PlacementListeners. PlacementEvent 518 is used to notify PlacementListener 514 if ViewController needs to be adjusted on the screen.

Next, TopListener 516 is an interface that performs specific desktop duties, such as, for example, the launching of an application, performing a desktop shutdown, displaying a message on the desktop, displaying a context sensitive title on the desktop or reconfiguring and non-intrusive observing the JTC application. TopEvent 520 is used to send a message to the TopListener to indicate that some action is needed on the desktop.

A desktop provides the operating-specific functionality of a windowing system and application management. For example, in Windows 95, the graphical interface that starts up is called a "desktop". The Windows 95 GUI itself is the "desktop system". Additionally, a desktop application provides a view or window in which an application may run and launch other applications. This is typically accomplished by a display of a hierarchical list of applications, which may be selected and "launched".

ApplicationMediator 512 may generate and receive RequestEvents, such as RequestEvent 522. A RequestEvent,

as used herein, represents a "lightweight transaction" that is used as an indication that an event and a service is required to process the event. A RequestEvent 522 is identified by major and minor codes. In addition, a
5 reference to a data model may be included for additional semantics. In turn, Transporter 524 in architectural pattern 500 is used to map RequestEvents, such as RequestEvent 522, to Destinations 528, 530, and 532. In other words, Transporter 524 acts as an event broadcast
10 mechanism within architectural pattern 500. Transporter 524 will route RequestEvents, such as RequestEvent 526 to various Destinations, such as Destination 528, Destination 530, or Destination 532. These destinations will interpret the RequestEvent 526, locate a server,
15 create a message format, create a protocol and deliver the information to the server's service for processing. These response data will be returned to the Transporter 524 in a RequestEvent, such as RequestEvent 526. In turn, Transporter 524 will return the RequestEvent to
20 ApplicationMediator 512, which will process the data contained in the event accordingly. For example, the return data may be sent to ViewController 502 to refresh the view displayed on the screen to the user. If an error occurs, a RequestException 534 may be generated and
25 returned to Transporter 524 and returned to the ApplicationMediator 512.

The ApplicationMediator 512 can fire RequestEvents to the Transporter synchronously, asynchronously and repeating asynchronously. A synchronous RequestEvent
30 will block the ApplicationMediator and either an error RequestException will be thrown or a possibly modified

RequestEvent will contain the results. An asynchronous RequestEvent will return immediately. The error RequestException or a possibly modified RequestEvent will later be "called back" to the ApplicationMediator at the requestException and requestResponse interface, respectively.

In depicted examples, a type Object data, such as Data 536 may be passed in reference by ViewController 502 to a Destination such as Destination 1 528. This data is passed via different events, such as ViewEvent 510, RequestEvent 522, and RequestEvent 526. Changed data, such as Data 538 may be returned from a Destination to ViewController 502 for display. This type Object data may take various forms, such as Extensible Markup Language (XML), String, Hypertext Markup Language (HTML), key/value, Remote Method Invocation (RMI), J/XFS, RS232, etc. The ViewController reads and modifies the Object data but does not create the data and the Destination creates, reads and modifies the Object data and sends it to and receives it from a server.

In addition, architectural pattern 500 also includes a Factory 540, which is used to allocate objects. Factory 542 is employed to also allocate singleton objects.

JTC 544 the system also contains a JTC interface implemented by all major objects in a JTC application including ApplicationMediator, ViewController, Destination and Transporter. Together, the objects implementing the JTC interface give the external appearance of a single application as they implement and propagate the required JTC interface methods.

The architectural pattern separates ViewController

502 (GUI view) from ApplicationMediator 512 (GUI state transition) from PlacementListener 514 (GUI placement) from RequestEvent 522 (lite transaction) creation from Destination 528 (transaction dispatch) from data creation
5 528, from data usage 502, and from transaction fulfillment.

The JTC interface of the present invention provides automatic non-intrusive tracing of application, and automatic non-intrusive event logging of application.

10 The ValidationRules provide class separation of chained field validation logic from both GUI and data model. Also the ValidationRules can run on client, server, or both.

The Destinations provide plugable data model,
15 network protocol, and message format modules for remote or local fulfillment. Further, Destinations can support both local and remote configurations in addition to multiple chains of them being called FIFO. The simpler the Destinations, the thinner the application. The more
20 complex the Destinations, the thicker the application.

The ApplicationMediators can be nested. PlacementListeners provide for partitioning of how to place components on the screen from what the components are and what order they arrive. RequestEvents
25 (lite transactions) can be broadcast to multiple servers using multiple network protocols with various data and message formats.

The architecture also provides permission keys to support intra and inter ViewController and
30 ApplicationMediator user ID and/or group enabling and disabling, and provides settable generic field level, focus level, and RequestEvent level ValidationRule

invocation.

The ViewControllerBaseImpl provides a mechanism to switch between various graphics containers. The example code described herein is illustrated in Java, but the processes of the present invention may be applied to other programming languages.

IV. Class Hierarchy

With reference now to Figure 6, a diagram illustrating classes in a class hierarchy is depicted in accordance with a preferred embodiment of the present invention. Class hierarchy diagram 600 illustrates the class hierarchy of new classes introduced by the present invention. All of the classes inherit from the class java.lang.Object. These classes provide an architectural pattern for client development in Java. The classes illustrated in class hierarchy diagram 600 contain interfaces and classes to support the architectural pattern of the present invention. Although the depicted examples are illustrated with respect to Java, the architectural pattern of the present invention may be applied to other types of programming environments. The depicted examples are not meant to be limitations on the programming environment in which the present invention may be applied.

With reference now to Figure 7, a unified modeling language diagram is depicted in accordance with a preferred embodiment of the present invention. Unified modeling language (UML) diagram 700 is a diagram illustrating the class hierarchy between various classes and interfaces within the architectural pattern of the present invention.

- UML diagram 700 uses the following conventions. A sub-interface of another interface (e.g., ViewController 702 to JTC 742) or a subclass of another class (e.g., ViewControllerImpl 704 to ViewControllerBaseImpl 706) is shown by a solid line from the subclass to the parent class where there is an open arrowhead. If a class is used to implement an interface, then a dashed line goes from the class to the arrowhead at the interface (e.g., ApplicationMediatorImpl 720 to ApplicationMediator 718).
10. A class or interface may be aggregated into another; this is indicated by a solid line connecting to a small diamond (e.g., ViewEvent 714 to ViewListener 718). The end with the small diamond represents the class that is aggregating (or containing) the class at the other end.
15. The multiplicity of the association is indicated by numbers at each end of the line, such as ViewEvent having multiplicity of 0 to many (with "many" indicated by "*") and ViewListener 718 having multiplicity one. So each instance of a ViewListener 718 may contain zero or more instances of ViewEvent 714.
- 20

Occasionally, alternative methods may be present for implementing an object. For example, ViewControllerBaseImpl 706 may implement a Container by either using the Java Abstract Window Toolkit (AWT) Panel 707 or a JFC 709. The small dashed lines indicate a choice.

25

Turning now to a discussion of the classes illustrated in Figures 6 and 7, ViewController 702 is an interface that defines an interface for a class that will be a single Component containing user interface components that are logically related to show information to the user and take input from the user. In particular,

30

ViewController 702 is a public interface that defines reusable graphics components as part of an overall client application.

With reference now to Figures 8A and 8B, diagrams illustrating variables and methods in ViewController 702 are depicted in accordance with a preferred embodiment of the present invention. The abstract ViewControllerImpl class implements the ViewController interface to provide default behavior where possible. The user can subclass ViewControllerImpl so that not all of the interface methods of ViewController need be implemented.

The ViewControllerImpl 704 is an abstract class implementation of ViewController. ViewControllerImpl 704 and ViewController 702 are the basic GUI building blocks for thin clients. ViewControllerImpl 704 provides the function that will be required by a subclass (not shown).

ViewControllerImpl 704 is controlled and mediated by an ApplicationMediator. ViewControllerImpl 704 extends the ViewControllerBaseImpl class and implements the ViewController interface so as to support the event handling methods and provide a default implementation for the other ViewController methods. This class maintains a reference to the client data model, the user validation level, and the UI properties and resources. Methods are present to add and remove listeners for view events generated, fire (send) view events to the appropriate listeners, cleanup list of event listeners on clear and exit, and enable or disable the panel.

With reference now to Figures 9A-9C, diagrams illustrating variables, constructors, and methods for ViewControllerImpl 704 are depicted in accordance with a preferred embodiment of the present invention. Table 900

in Figure 9A illustrates the variables used in
 ViewControllerImpl 704. The tables used to describe the
 class include a name of the variable, constructor, or
 method, a declaration for the variable, and a description
 5 for the variables. Table 902 in Figure 9B contains a
 similar table for the constructor used in
 ViewControllerImpl 704. Table 904 in Figure 9C provides
 a list of the methods used within ViewControllerImpl 704.
 Similarly, table 904 includes a name for the method, a
 10 declaration used for the method, and a description of the
 method.

ViewControllerImpl 704 is usually a Java Component
 or Container or bean with the extra methods specified
 here. The basic operation of a ViewControllerImpl is
 15 handles using the following steps: (1) implement the
 ViewController and JTC interfaces; (2) add your bean
 specific methods; (3) create and compose GUI; (4) return
 the Component in getComponent(); (5) return and set
 permission keys, resources and properties when asked; (6)
 20 update permission keys, resources and properties when
 asked; (7) handle internal AWT events; (8) validate and
 format fields with ValidationRules; (9) issue ViewEvents
 for semantic interpretation; (10) return to the AWT
 Thread as soon as possible; (11) handle refresh(data)
 25 calls by updating GUI; (12) access data; and and (13)
 repeat from step (3).

ViewControllerImpl 704 is controlled and mediated by
 an ApplicationMediator. The default function provided by
 this class includes: from ViewController interface:
 30 add/remove ViewListener, fireViewEvent,
 setValidationLevel, refresh, getComponent, and AWTEvent
 threading. From the JTC interface, the functions

provided include: clear, exit, set/getEnabled, set/getVisible, and toString.

ViewControllers know the full semantics of the data model being used, even though only a generic *Object data*; type reference is provided. The ViewControllers will cast generic reference into Class based data objects. Some design principles to use in ViewControllers when deciding how to communicate information to ViewListeners are described below.

10 A ViewEvent is created, its setter methods are called, and it is fired to the ViewListeners. The listeners interpret the major and minor codes to decided what to do.

To provide additional information, the
15 ViewController can set a Java data object in the ViewEvent using the setData method. The data reference can be simple String objects or user defined data objects.

A similar approach to provide additional information
20 is for the ViewController to supply application defined objects using the setData method. The data reference are more complex than simple objects (i.e. Customer, AccountInfo, Schematic, CarSpec, etc.).

The ViewController also can provide named methods.
25 For example, a ViewListener may call customerVC.getCustomerName() to retrieve the customer name. The implementation of getCustomerName is hidden by the ViewController and could involve manipulating several Java Components. In addition, setters may also be
30 provided, such as VC.setCurrentCustomer(). In general, too many named methods may suggest that information sharing should be via data objects within ViewEvents and

refresh methods instead. The use of named methods may reduce reuse of a ViewController by an ApplicationMediator.

The ViewController may listen to property change events on a data model using standard Java mechanisms. Thus, the ViewListeners, the ViewControllers, or any other interested object can communicate via the data model changes. In general, this is very data-oriented object model programming approach and can lead to object-spaghetti code. Changes in the data model ripple throughout the application.

ViewControllerBaseImpl 706 is an abstract class and is a superclass of ViewControllerImpl 704. This class provides the indirection to allow specification of the Component type of the ViewController via a "has a" and/or "is a" relationship. The idea is for the developer to replace this class entirely with the developer's own implementation. This replacement may be accomplished by creating the developer's own implementation of ViewControllerBaseImpl that implements the methods getComponent(), setEnabled(boolean enable), and setVisible(boolean visible). ViewControllerBaseImpl 706 is generally used with either JPanel 707 or a java.awt.Panel 707 by changing the inherits statement. ViewControllerBaseImpl may also be used with any Java or user-defined component or container. The variables, constructors, and methods for ViewControllerBaseImpl 706 are found in Figures 10A-10C. The variable for ViewControllerBaseImpl 706 is found in table 1000 in Figure 10A while the constructors are listed in table 1002 in Figure 10B. The methods are listed in table 1004 in Figure 10C.

ViewControllerAdapter 708 is an abstract class and is a helper adapter class that fits between ViewControllerImpl 704 and the developer's subclass. ViewControllerImpl 704 implements almost all of the standard Java AWT listeners and provides empty methods. This class allows ViewControllers to be implemented without having to specify all of the methods of a Java AWT listener interface when a subclass being created needs only one such method.

10. ViewControllerAdapter 708 implements the following AWT event listeners: AdjustmentListener, ComponentListener, FocusListener, ItemListener, KeyListener, MouseListener, TextListener. This class knows about everything that ViewControllerImpl knows, and also knows the default empty implementation of AWT event listener methods. The methods on the class allow for adding and removing of listeners for view events generated, firing (sending) of view events to the appropriate listeners, cleaning up the list of event listeners on clear and exit, and enabling or disabling the panel.

Figures 11A-11C illustrate a variable, a constructor, and methods for ViewControllerAdapter 708. Table 1100 in Figure 11A illustrates the variable while table 1102 in Figure 11B illustrates the constructor of ViewControllerAdapter 708. Methods for ViewControllerAdapter 708 are shown in table 1104 and Figure 11C.

30. ValidationRule 710 is an abstract class for class-based value validation rules used to validate and format ViewController contents. Validation is normally used for single or groups of text fields (user entry).

Examples include range checking, date, social security number, or other business specific formats. The definition of the validation rules are kept out of the data models and ViewControllers to maximize reuse of validation logic. These types of objects only encode which rule to use.

Typically, validation rules have two methods: (1) edit and (2) normalize. The edit method is used to input a string, validate the value and output the view friendly formatted string. The normalize method is used to input a string, validate the value and output a string that will be set into the data model and/or transmitted via a RequestEvent to a destination. For example, given a value of "1000.0", edit will produce "\$1,000.00" and normalize will produce "1000". The ValidationRule provides two static helper functions to apply edits or normalizes on an input string given a list of edit rule class names. This is called ValidationRule chaining. In this manner, ValidationRule 710 allows a developer to keep the implementations of validation rules small and simple so as to maximize reuse of validation code. Additionally, ValidationRule 710 may be run on a server. The methods can create an instance of a ValidationRule given a class name, apply edit rules sequentially on a string given a list of ValidationRule class names, or apply normalize rules sequentially on a string given a list of ValidationRule class names. Invocation of a validation rule is initiated by the ViewController or some business logic method. In case of failure, a ValidationRuleException is thrown.

With reference now to Figures 12A-12C, drawings illustrating variables, constructors, and methods for

ValidationRule 710 are depicted in accordance with a preferred embodiment of the present invention. The variables are illustrated in table 1200 in Figure 12A while the constructor is shown in table 1202 in Figure 12B for ValidationRule 710. The methods used in ValidationRule 710 are shown in table 1204 in Figure 12C. Figure 12D contains code 1206, which is used to apply ValidationRules for classes in response to ValidationRule 710 being given a list of class names for processing). Code 1206 will apply each ValidationRule for a class and return a formatted result. A similar code is used for ValidationRule.applyNormalizes.

ValidationRuleException 712 is a class that extends the Exception class in Java to represent the type of exceptions thrown by rules defined in ValidationRule 710. ValidationRuleException 712 is generated when a validation rule has failed. The cause of the failure is contained in the exception. In the depicted examples, processing of remaining validation rules will halt. This class needs to know the validation rule exception string.

With reference now to Figures 13A and 13B, tables illustrating variables and constructors for a ValidationRuleException are depicted in accordance with a preferred embodiment of the present invention. Table 1300 in Figure 13A illustrates a variable while table 1302 in Figure 13B illustrates constructors for ValidationRuleException 712.

ViewEvent 714 is a class having a mechanism used for communication between ViewControllers and ViewListeners, such as ApplicationMediators and between ApplicationMediators. Notification occurs when a

significant AWTEvent has occurred that the ViewController cannot handle. Both ViewControllers and ApplicationMediators can fire ViewEvents. The ViewEvent state includes a major code, a minor code, a source
5 Object of the event, and a generic Object data reference. Numerous predefined major codes and minor codes are provided and a subclass can define additional application specific codes.

When a ViewController fires a ViewEvent, it is
10 handled in the ApplicationMediatorImpl superclass first and then forwarded to the ApplicationMediatorImpl subclass in the processViewEvent method. This mechanism allows threading and queuing to be handled by the superclass. Dispatching a ViewEvent to a subclass is
15 performed in one of two ways: queued-event dispatching and thread-event dispatching.

With reference now to Figures 14A-14D, diagrams illustrating variables, constructors, and methods for ViewEvent 714 are depicted in accordance with a preferred
20 embodiment of the present invention. Table 1400 in Figures 14A-14D illustrates variables for ViewEvent 714. Table 1402 in Figure 14E shows the constructors for ViewEvent 714. The methods for ViewEvent 714 are shown in table 1404 in Figure 14F.

25 ViewListener 716 is an interface for receiving ViewEvents generated by ViewControllers. The listener will be called back on the viewEventPerformed method. ViewListener 716 interprets the values from the ViewEvent and decides what action to take. If more information
30 needs to be conveyed, ViewListener 716 gets the data variable. A ViewListener can set the ViewEvent to be consumed. When the ViewEvent is consumed, the ViewEvent

will not be forwarded to other ViewListeners.

Turning now to Figures 15A and 15B, diagrams illustrating a variable and a method for ViewListener 716 are depicted in accordance with a preferred embodiment of the present invention. The variable for ViewListener 716 is shown in table 1500 in Figure 15A. The method used in ViewListener 716 is shown in table 1502 in Figure 15B.

ApplicationMediator 718 is an interface that specifies methods for the management of PlacementListeners, ViewListeners, RequestListeners, TopListeners, permissions, properties, resources, visibility, validity. ApplicationMediator 718 also defines a generic method to refresh data. ApplicationMediator 718 defines the interface for a class that will mediate multiple ViewControllers or other ApplicationMediators. Methods are provided in this class to add and remove PlacementListeners for the ViewControllers, to add and remove TopListeners for the ApplicationMediator, to add and remove RequestListeners for RequestEvents generated, and to add and remove ViewListeners for ViewEvents passed up from the ViewControllers. Other methods in the ApplicationMediator 718 set the data used by the ViewControllers, tell where and how to handle ViewEvents generated by ViewControllers, and generate RequestEvents based on some application logic of ViewEvents. Methods are provided in this interface to set the property information and the resources used by the ViewControllers.

The Figures 16A and 16B, diagrams illustrating a variable and methods for ApplicationMediator 718 are depicted in accordance with a preferred embodiment of the

present invention. Table 1600 in Figure 16A illustrates the variable used for ApplicationMediator 718 while table 1602 in Figure 16B illustrates the methods used in ApplicationMediator 718.

5 ApplicationMediatorImpl 720 is an abstract class, implements the ApplicationMediator interface, and provides default behavior to manage ViewControllers, ApplicationMediators, add/fire/remove of PlacementListeners, RequestListeners, ViewListeners and
10 TopListeners. A subclass of ApplicationMediatorImpl 718 should focus on managing the state machine for ordering the ViewControllers it creates and the mediating of events. This focus includes determining which ViewController to allocate, which ViewController to make
15 visible, when to fire a PlacementEvent, when to fire a RequestEvent, and when to fire a TopEvent. A default list is provided to hold ViewControllers. The default list is a Vector and is directly available to subclasses. An ApplicationMediatorImpl 720 may create other
20 ApplicationMediators. As a result, a list to hold ApplicationMediators also is provided.

 The helper methods initApplicationMediators and initViewControllers will load their respective classes and store them in Vectors. The helper methods getAM,
25 getVC, setAM, and setVC provide a way to get/set the elements in the Vectors.

 When a ViewController fires a ViewEvent, the ViewEvent is handled in ApplicationMediatorImpl 720 first and then forwarded to the subclass (not shown) in the
30 processViewEvent method. This process allows threading and queuing to be handled first by ApplicationMediatorImpl 720.

ApplicationMediatorImpl 720 needs to have reference to the client data model, PlacementListeners, TopListeners, ViewListeners, ViewControllers, other ApplicationMediators, UI properties and resources, and

5 Event processing threads.

Methods are provided in ApplicationMediatorImpl 720 to add and remove listeners for ViewEvents generated or forwarded from a ViewController, to fire (send) ViewEvents to the appropriate listeners, to add and

10 remove listeners for RequestEvents generated, and to fire TopEvents and to fire (send) synchronous or asynchronous RequestEvents to the appropriate listeners. Other methods add and remove listeners for PlacementEvents generated, fire (send) PlacementEvents to the appropriate

15 listeners, and cleanup a list of event listeners and event processing threads on clear and exit. It is also possible to enable or disable the visibility, create, initialize, and manage ViewControllers, and to create, initialize, and manage other ApplicationMediators. Other

20 methods get references of all JTC objects managed, set client data references for all ViewControllers and ApplicationMediators managed, add TopListeners, and handle the processing of a received ViewEvent in a separate thread (either in a queue fashion or unique

25 thread per event).

Turning next to Figure 17A-17D, diagrams illustrating variables and a constructor for ApplicationMediatorImpl 720 are depicted in accordance with a preferred embodiment of the present invention.

30 Table 1700 in Figure 17A illustrates variables for ApplicationMediatorImpl 720 while table 1702 in Figure 17B shows the constructor used in ApplicationMediatorImpl

720. Table 1704 in Figures 17C and 17D illustrate the methods used in ApplicationMediatorImpl 720.

With reference now to Figures 17E-17H, diagrams illustrating code used in methods for ApplicationMediatorImpl 720 are depicted in accordance with a preferred embodiment of the present invention. In Figure 17E, code 1706 illustrates code used in an exit method for ApplicationMediatorImpl 720. This code allows the ApplicationMediator to be exited by exiting all allocated ViewControllers and ApplicationMediators. All data will be set to null by code 1706 and all lists destroyed.

In Figure 17F, code 1708 illustrates code used to clear an ApplicationMediator by clearing all allocated ViewControllers and ApplicationMediators. The data is set to null, but the lists are not destroyed. A cleared ApplicationMediator can be used again. In Figure 17G, code 1710 is used to initialize an ApplicationMediator using the listeners of an existing ApplicationMediator. In Figure 17H, code 1712 is used to refresh an object. This code is used to refresh ViewControllers and ApplicationMediators when new data arrives. In Figures 17I-17L, depicts code 1714 which is used to handle the event dispatch.

PlacementEvent 722 is a class used to notify PlacementListeners that a ViewController needs to be adjusted on the screen. The value stored in a PlacementEvent 722 are: (1) major code indicating the placement type; (2) minor containing additional information; (3) source, which indicates the sender of the event; (4) Component, which is the ViewController's

component; and (5) data, which is a generic Object data, containing the type to pass more information and is typically not the same type of data sent in ViewEvents and RequestEvents. Predefined placement types, which are
5 used as major codes include, for example, add, remove, and modify. A subclass may define more application specific PlacementEvent codes. Additional information may be conveyed through the use of the minor variable or by supplying real data.

10 With reference now to Figures 18A-18C, diagrams illustrating variables, constructors, and methods for a PlacementEvent are depicted in accordance with a preferred embodiment of the present invention. Table 1800 in Figure 18A illustrates variables used within
15 PlacementEvent 722 while table 1802 in Figure 18B show constructors used for PlacementEvent 722. Table 1804 in Figure 18C shows the methods used within PlacementEvent 722.

PlacementListener 724 is an interface used to manage
20 the placement of ViewControllers on the screen. For a given component or container, such as a ViewController, PlacementListener 724 controls the placement while ApplicationMediator controls the ordering of ViewControllers instead of their placement. For example,
25 if a designer wishes to create a split pane with a tree in the left pane that allows selection of ApplicationMediators, ViewControllers will display on the right. As a result, an "add" method is called to the split pane. If a frame is used, "add" has a different
30 syntax. An object handling placement of components will register as a listener for notifications to place objects on the screen. This role is the one played by

PlacementListener 724. The ApplicationMediator controls the ordering of ViewControllers, not their placement. A PlacementListener adds itself to a PlacementEvent generator and is called back with a PlacementEvent which indicates a desired action. This mechanism allows complete separation between view creation (ViewController), view ordering (ApplicationMediator), and view placement (PlacementListener). This arrangement increases the reusability of ViewControllers and ApplicationMediators to nearly 100% with respect to placement.

With reference now to Figures 19A and 19B, diagrams illustrating a variable and method for a PlacementListener are depicted in accordance with a preferred embodiment of the present invention. Table 1900 in Figure 19A illustrates the variable used in PlacementListener 724 while table 1902 in Figure 19B illustrates the method used in PlacementListener 724.

TopEvent 726 is a class used to notify TopListeners that ApplicationMediator needs some desktop function to occur. The value stored in TopEvent include: (1) major, which identifies the TopEvent type; (2) minor, which contains additional information; (3) source, which identifies the sender of the Event; (4) consume, which will result in the Event being processed if true; and (5) data, which is a generic object data used to pass more information. Predefined types, which are used as major codes include, for example, EXEC, BROWSER, , TITLE, STATUS, and OS. This category of types is used for interaction with the desktop (operating system, browser, other applications). Additional major codes include TRANSPORTER, DESTINATION, APPLICATION_MEDIATOR, ,

REQUEST_EVENT, VIEW_CONTROLLER, VIEW_EVENT, TOP_LISTENER, TOP_EVENT, PLACEMENT_LISTENER, PLACEMENT_EVENT, MAJOR, MINOR, AWT_EVENT, and JTC. This category of types is used to indicate a reconfiguration of the JTC

5 application. Additional TopEvent codes may be defined through the use of subclasses. Additional information may be conveyed by using the minor variable or supplying real data.

With reference now to Figures 20A-20C, diagrams
10 illustrating variables, constructors, and methods for a TopEvent are depicted in accordance with a preferred embodiment of the present invention.

Table 2000 in Figure 20A illustrates the variables contained in <update this table based on latest javadoc
15 contents> TopEvent 726 while table 2002 in Figure 20B shows the constructors for TopEvent 726. The methods for TopEvent 726 are found in table 2004 in Figure 20C.

TopListener 728 is an interface that performs business specific desktop duties for a display.
20 TopListener 728 provides an interface that is highly customizable for each business application. This interface specifies a topEventPerformed method to receive a TopEvent encoding a task to be performed.

With reference now to Figure 21A and 21B, diagrams
25 illustrating a variable and methods for TopListeners are depicted in accordance with a preferred embodiment of the present invention. Table 2100 in Figure 21A illustrates the variable for TopListener 728 while table 2102 in Figure 21B illustrates the methods in TopListener 728.

30 The RequestEvent 730 is a class that extends EventObject to represent RequestEvents generated by ApplicationMediators. This class represents a

lite-weight transaction. The instances of this class denotes a unit of work. RequestEvent 730 represents a request for a transaction or data from a source external to the application. The request is handled by a remote or local destination. The Transporter is responsible for mapping RequestEvents to Destinations. The state of a RequestEvent 730 includes the following information: (1) major code identifying a family of RequestsEvents; (2) minor code identifying a specific request under the family); (3) version, which is usually user defined; (4) status which may be an appended string showing the stages of processing; (5) consume (a consumed Request Event will cause JTC to stop processing the Request Event); and (6) data, which may be, for example a reference to generic Object data. A RequestEvent can be set to be consumed in which case Destinations and the Transporter will halt forwarding of the RequestEvent and return control to the ApplicationMediator.

Turning now to Figures 22A-22C, diagrams illustrating a variable, constructors, and methods for RequestEvent are depicted in accordance with a preferred embodiment of the present invention. Table 2200 in Figure 22A illustrates the variable in RequestEvent 730. Table 2202 in Figure 22B shows the constructors used in RequestEvent 730. Table 2204 in Figure 22C illustrates the methods in RequestEvent 730.

RequestException 732 is a class that extends Exception to represent the type of exception thrown when there is a problem in submitting a RequestEvent. This class is used when the processing of a RequestEvent causes an error condition. This exception can be thrown directly when a synchronous or asynchronous RequestEvent

is performed. This indicates failure in synchronous processing or failure in the fire mechanism. The exception can be thrown indirectly when asynchronous RequestEvent is processed. The cause of the failure is also indicated in the exception in the depicted examples. Thus, this class needs to know the string representing the reason for the RequestEvent exception.

With reference now to Figures 23A-23C, diagrams illustrating a variable, constructors, and methods for RequestException 732 are depicted in accordance with a preferred embodiment of the present invention. Table 2300 illustrates the variable for RequestException 730 while table 2302 in Figure 23B illustrates the constructors used in RequestException 732. The methods for RequestException 730 are found in table 2304.

The RequestListener 734 is an interface that defines the method for a class to implement for receiving synchronous or asynchronous RequestEvents. This interface is implemented by classes that want to listen and be notified when a RequestEvent is fired. A typical example is a Transporter instance. The RequestListener is added to an ApplicationMediator as a listener for RequestEvents. Two styles of notification are possible: synchronous and asynchronous. The Transporter implements the RequestListener interface to map RequestEvents to Destinations. Other classes can implement the RequestListener interface and register for RequestEvents to monitor them. A RequestException can be thrown directly when invoking synchronous and asynchronous RequestEvents or on a callback when invoking only asynchronous RequestEvents.

With reference now to Figures 24A and 24B, diagrams

illustrating a variable and methods for a RequestListener are depicted in accordance with a preferred embodiment of the present invention. Table 2400 in Figure 24A

illustrates the variable in RequestListener 734 while
5 table 2402 in Figure 24B illustrates the methods used in RequestListener 734.

RequestResponseListener 736 interface is implemented by a class that requires asynchronous RequestEvent processing. For example, when an ApplicationMediator
10 implements this interface, it will be called back on one of the two methods after it fires asynchronous RequestEvents. Successful RequestEvents are returned through the requestResponse method and passed a RequestEvent that may have new data. Unsuccessful
15 RequestEvents are returned through the requestException method and the RequestException contains the reason.

Turning now to Figures 25A and 25B, diagrams illustrating a variable and methods for a RequestResponseListener are depicted in accordance with a
20 preferred embodiment of the present invention. Table 2500 in Figure 25A shows the variable in RequestResponseListener 734 while table 2502 in Figure 25B illustrates the methods in RequestResponseListener 736.

25 The Transporter 730 is a class that implements the JTC and RequestListener interfaces. The primary function of Transporter class 730 is to map RequestEvents to Destinations. In other words, Transporter 738 acts as an event broadcaster to route RequestEvents to the
30 appropriate Destination. Typically, ApplicationMediators fire RequestEvents which are routed to a Destination by

the Transporter. The Destination processes the RequestEvent by performing some interpretation and sending them to a target, such as a server. This event broadcasting is performed by Transporter 738 based on the major code of the RequestEvent in the depicted examples. Transporter 736 needs to know the Destinations, the major codes registered by the Destinations, RequestEvents, RequestResponseListener, and RequestExceptions.

Methods are provided in Transporter 738 to register a Destination that processes RequestEvents with a specific major code, remove the registration of a Destination, receive a RequestEvent for synchronous or asynchronous processing, handle the processing of a received asynchronous RequestEvent in a separate thread (either in a queue fashion or unique thread per event), and return a list of Destinations registered for a specific key (major code). Other methods cancel the processing of an asynchronous RequestEvent, cleanup threads and lists of Destinations on clear and exit, enable or disable the sending of RequestEvents to Destinations, and enable or disable the tagging (status setting) of RequestEvents.

With reference now to Figures 26A-26C, diagrams illustrating variables, a constructor, and methods for a transporter are depicted in accordance with a preferred embodiment of the present invention. Table 2600 in Figure 26A illustrates variables for Transporter 736 while table 2602 in Figure 26B illustrates a constructor for Transporter 738. The methods for Transporter 738 are illustrated in table 2604 in Figure 26C.

Turning now to Figures 26D-26E, diagrams illustrating code used in methods in Transporter 738 are

depicted in accordance with a preferred embodiment of the present invention. Code 2604 in Figure 26D illustrates processing or selecting Destinations in response to a RequestEvent. Given a RequestEvent of Destinations, code 5 2604 will call each Destination in a first in first out (FIFO) / first exception first return (FEFR) order. Code 2606 in Figure 26E illustrates a process for submitting a synchronous request. For each Destination listening for the current family of RequestEvents, as indicated by the 10 major code, the RequestEvent is sent to a Destination for processing. If a problem occurs, a RequestException is thrown. Processing the RequestEvent will occur as long as a RequestException being thrown is not from a Destination and the RequestEvent is not consumed. 15 Submission of an asynchronous request is performed by code 2608 in Figure 26F. Code 2610 in Figure 26G illustrates code for handling executions of submits on another thread.

The Destination 740 is an interface used to pass 20 results or events. This interface is implemented by classes that want to listen to Transporters and send RequestEvents to servers. Destination 740 performs the following functions: (1) server: locate it; (2) network transports: create the low level protocols such as 25 sockets, RMI, URLs, Files, etc.; (3) server types: be able to talk to different servers such as EJBs, Servlets, web servers, legacy, etc.; (4) message formats: be able to turn RequestEvents and data into formats that the servers understand; (5) timeouts: finish the job in 30 the appropriate amount of time or cancel the operation; (6) retries: how many times should a connection be tried; (7) caching: save data until it becomes stale;

(8) stale data: detect stale data; (9) heart beats: I am alive; (10) logging: save pre and post RequestEvents; and (11) objects: locate them, create them.

Example Destinations behaviors include the following: (1) serializing a request and writing it to a socket, (2) reading and writing requests and responses from a file for demonstrations, (3) casting the data in a RequestEvent into objects and so that RMI methods are visible, (4) debugging, (5) tracing, (6) compression, (7) encryption, and (8) translation.

A Destination 740 is first added to a Transporter as a RequestListener, listening for RequestEvents of a specific major code. When RequestEvents enter the Transporter, these RequestEvents are sent to each listening Destination via a requestEventPerformed. Additionally, a Destination monitors the state of the consumed attribute of the RequestEvent and will terminate processing when the state is true.

RequestEvents are identified by a major code (represents a family of Requests) and a minor code (represents a specific Request). Destinations are added to the Transporter as DestinationListeners specifying a major code for RequestEvents they are interested in receiving. The Destination is called when the major code of the RequestEvent matches the Destination's major code. Multiple Destinations can listen for the same RequestEvent major code and results of one Destination can be passed to the next Destination. RequestEvents are processed in a first in first out (FIFO) / first exception first return (FEFR) order.

With reference now to Figures 27A and 27B are diagrams illustrating a variable and methods for a

Destination in accordance with a preferred embodiment of the present invention. Table 2700 in Figure 27A illustrates the variable in Destination 740 while table 2702 in Figure 27B illustrates the methods used in
5 Destination 740.

DestinationImpl 742 is an abstract class that implements the Destination interface and provides a default implementation for the Destination interface methods. It needs to know whether the Destination is
10 enabled or disabled, whether the RequestEvents are tagged with a status or not, and the RequestEvent processing timeout value. Methods are provided to enable or disable the processing of RequestEvents in the Destination and to set the RequestEvent processing timeout value.

15 With reference now to Figures 28A-28C, diagrams illustrating variables, constructors, and methods for a DestinationImpl are depicted in accordance with a preferred embodiment of the present invention. Table 2800 in Figure 28A illustrates the variable used in
20 DestinationImpl 742 while table 2802 in Figure 28B illustrates the constructor used in DestinationImpl 742. Methods for DestinationImpl 742 are illustrated in table 2804 in Figure 28C.

In Figure 28D, code 2806 is used to process a
25 RequestEvent. If an application uses a set of objects that perform the retrieval and update of data, then those objects can be accessed from a generic Destination class. Typically, a Destination implementation accesses a single interface class with methods and is written specifically
30 to serve one purpose (i.e. the retrieval and update of customer data or the retrieval and update of employee data). However, a generic Destination class can be

written to retrieve and update any set of data. The requirement is that the class/object that serves the data must all have methods with the same interface (in other words, they must take in the same number and type of

5 object as parameters and return the same type or generic object.) This way, the Destination class can be implemented to use Java's reflection to access/create the object and invoke a particular method on the object. The major code of the RequestEvent can specify the class name and the minor code can specify the method name. The code
10 in Figure 28D shows how a Destination can be implemented to access an Enterprise JavaBean session bean instance from an application server using Java's RMI and Java's Reflection capabilities. The methods on the session
15 beans all take a single parameter of type Object and return a parameter of type Object. The client user interface and the session beans know the specific type of data objects passed but the Destinations do not need to know about the data.

20 JTC 744 is an interface that provides a top level interface in the architecture of the present invention. All major objects in a JTC application implement this interface including the driver programs that launch the application/applet. This interface provides the ability
25 to reference all objects using a single type. The behaviors expected include clear, exit, getJTCs, init, isEnabled, setEnabled, and toString. The interfaces that extending JTC are ViewController, ApplicationMediator, and Destination. The classes
30 implementing JTC are ViewControllerImpl, ApplicationMediatorImpl, Transporter, and DestinationImpl.

Of particular importance is the `getJTCs()` method. Each JTC object returns the other JTC objects it creates. Iteration over these objects is possible. Inspection of the object types and addition of the appropriate
5 listeners may be performed. This interface provides a mechanism for non-intrusive logging, tracing and debugging.

Factory 746 is a class used to make objects. This class is the place for code relating to prefetching,
10 caching, and alternative "styles" for the new operator (or construction of objects), or loading a serialized object. The alternative "style" may be, for example, using `Class.forName`. The default behavior of a Factory includes, for example, creating a single object.
15 Additionally, the behavior may include returning a previously created object if a key has been used or otherwise creating a new object and remembering the object.

Further, Factory 746 may be used to create multiple
20 objects. Also, if a key[i] has been used, a previously created object may be returned. Otherwise, a new object [i] may be created and remembered by the Factory. For example, on a single JVM, a designer may want to create a single shared instance of a `Transporter`. Each access to
25 the `Transporter` should be as follows: `Transporter t = Factory.newInstance(com.ibm.jtc.Transporter, true)`. In this case, the `Class.forName` method is used with the key to create an instance of the `Transporter`. The true value indicates that a singleton may be created. The singleton
30 may be turned off in which case all objects created are new and not remembered. Another key also may be used besides the class name. An example is as follows:

Customer c = Factory.newInstance ("CUST",
com.abc.Customer, true). The user can create a single
object or create multiple of objects. For example, on a
single JVM, a user may want to create and share one
5 instance of a Transporter. The Factory class needs to
know the list of singleton classes managed. The methods
in this class provide several ways to create an instance
of a class given one or more full path class name
strings. The Factory class also can remove stored
10 references of singleton classes.

With reference now to **Figures 29A and 29B**, diagrams
illustrating variables and methods in a Factory are
depicted in accordance with a preferred embodiment of the
present invention. Table 2900 in **Figure 29A** illustrates
15 the variable used in Factory 746 while table 2902 in
Figure 29B illustrates the methods for Factory 744.

Q 740 is a class representing a queue of objects
that are added and removed in first-in first-out (FIFO)
ordering. The size of the queue can be set to a maximum
20 or as unbounded. The structure maintains the queue of
objects, the current size of queue, and the front and
back ends of the queue. Methods are provided to add an
object to the back end of the queue, remove an object
from the front of the queue, and predicates to check if
25 the queue is empty or full.

JTC 744 is a top level interface, which all major
objects in an application created using the architecture
of the present invention should implement. This would
include driver programs that launch the application.
30 This interface allows referencing of objects with a
consistent type and a similar expected behavior.

Of particular importance in this class is the

getJTCs() method. Each JTC object returns the other JTC objects it creates. With this interface, a process can iterate over these objects, inspect their types, and add the appropriate listeners. The system may be reconfigured through this interface. This reconfiguration includes, for example, disabling the Transporter, adding a priority Destination, and consuming an event. This interface provides an excellent mechanism for non-intrusive logging, tracing and debugging. For example, if a program Test1.java implements JTC, it will return the Transporter, ApplicationMediators and Destinations it creates.

With reference now to Figure 30A and 30B, tables illustrating variables and methods in a JTC is depicted in accordance with a preferred embodiment of the present invention. Table 3000 in Figure 30A illustrates variables in JTC while table 3002 in Figure 30B illustrates methods for JTC.

V. Steps In Building An Application Using The JTC Architecture

A thin client application or a thick client application that follows the JTC architecture of the present invention can be built using a top-down, bottoms-up, or from the middle approach. All approaches allow for concurrent development within the JTC client application. All approaches allow for concurrent development of the JTC client application and the server and services.

The following steps show a bottoms-up process. A proper design of interfaces and subsystem models, will ensure that the JTC application can be implemented in

parallel.

A. Design Client Application Subsystems

The key to the successful implementation of a JTC
5 application that has reusable and maintainable parts is
the proper division of the application into logical
subsystems. This division should be driven by the
analysis of the application's domain. For example, using
an object-oriented analysis of the application's domain,
10 sets of use cases can be developed in the following
manner. The actors external to the system are
identified. For example, in designing an automated
teller machine (ATM) the customer using the machine would
be an actor and the bank's central computer is also an
15 actor. The use case is a set of interactions the actor
or actors have with the system. All sequences of
interactions, including normal and exceptional behavior,
should be specified. For example, normal behavior for
withdrawing cash from an ATM would include a sequence of
20 prompts and customer responses that eventually lead to
the dispensing of cash. Exceptions would include an
unreadable card, an incorrect PIN, insufficient cash in
the machine, insufficient funds in the account, etc. Use
cases can include a set of preconditions and
25 post-conditions. A precondition for the ATM would be
possession of an ATM card and a post condition, if the
cash withdrawal is successful, would be an appropriate
debit from the account balance.

The use cases produce natural divisions of function
30 in an application. Use cases describe functions of the
application that are most likely to be reused within and
outside of the application. Each use case or possibly a

group of fine-grained related use cases should make up the logical subsystem model. Groups of ViewControllers and ApplicationMediators are implemented in these subsystems. A ViewController is used to represent a
5 single reusable screen or grouping of information to be inputted and/or displayed. An ApplicationMediator mediates multiple ViewControllers for a single application function or use case.

Other subsystems of the client application include a
10 communications subsystem, a client startup subsystem, reusable graphical components, business validation rules, and enterprise policies.

The business logic and central data management of an application should be separated out from the JTC
15 application. This business logic and data can be physically located on any machine. JTC clients typically do not keep or manage transaction states; server side business logic manages the transaction states. Therefore, the communications subsystem is responsible
20 for the sending of data requests and transaction requests to the business logic outside of the application. This consists of the implementation of RequestEvents and Destinations.

A client startup subsystem manages the client
25 application lifecycle and overall look and feel. This subsystem includes implementations of TopListeners and PlacementListeners.

Common reusable graphical components used by user interface panels should be separated out in their own
30 subsystem so that they are designed and implemented for reuse and available to other subsystems and applications.

The rules for validating user input for display

format and persistent storage should be separated into another subsystem. This allows for central maintainability of the rules and reuse across all other subsystems and applications. ValidationRules are
5 implemented in this subsystem.

Enterprise policies include functions related to security, login/logout, users/groups/roles, profiles, locale, and languages. These requirements cross subsystems in a consistent and easily managed manner
10 through methods including set/getProperties, set/getResources, set/getPermissions. Additional non functional enterprise policies such as caching, data pre-fetching and mobile users are implemented in common easily configurable Destination subsystems.

15 In a bottoms-up approach, the client data model should be developed next for the specific subsystem. This model specifies the structure and contents of data used by the client application. There are a variety of approaches to a client data model: Local Object Model,
20 Workflow Object Model, XML, Named Methods, Key/Value Pairs, Ordinal Positioning, RMI, etc. To optimized concurrent development with the server and services, a bootstrap Key/Value data model is a sufficient and expeditious choice.

25 Once a data model approach is chosen, then the data objects must be created. These can be grouped and organized at different levels of granularity. In general, each component, such as, for example, a ViewController or an ApplicationMediator, should only
30 reference the data the component uses or manages. For example, a ViewController should only have data required to be displayed on a single screen and the data input by

the user on that screen.

B. Create a ViewController

In these examples, a ViewController is a panel that contains one reusable screen of user display and input.

5 This screen can either contain Java Abstract Windowing Toolkit (AWT) components or Java Foundation Classes (JFC/Swing) components. The way to choose between the two types of components is to implement the ViewControllerBaseImpl class. The ViewControllerBaseImpl

10 class can extend AWT's Panel class or JFC's JPanel class. Also ViewControllerBaseImpl can just contain a component, container or bean and return it in the getComponent() method. In a graphical user interface, components are organized into manageable groups through

15 the use of containers. Additionally, a container may provide basic window and dialogue services. A java.awt.Panel itself is a pure container. A java.awt.Panel itself is not a window in itself, but its sole purpose is to organize components in a window. With

20 reference now to Figure 31, a flowchart of a process for creating a ViewController is depicted in accordance with a preferred embodiment of the present invention. A particular screen is selected to implement (step 3100) and a class is created that extends one of the

25 ViewController implementation classes (step 3102). This class should have a suffix of VC to distinguish it as a ViewController class. The init() method is overridden (step 3104) and replaced with code to create, initialize, and layout graphical components of the panel. Visual

30 builders (such as IBM VisualAge) can be used to create this method automatically. The refresh() method is

overridden (step 3106). In this method, data to be displayed will be passed in. ViewControllers should be implemented to be reusable with different sets of data. By invoking refresh(), ApplicationMediators can control
5 for what purpose the ViewController is used. The clear() method is overridden (step 3108). In this method, the set of data used is cleared from use and display. This allows for the ViewController to be used with another set of data.

10. As part of component initialization, the ViewController is added as its component's event listeners (step 3110). The event listeners are implemented for the components (step 3112). ValidationRules are used to validate user input from a
15 component event (step 3114). Use ViewEvents (described below) to send user inputted data or requests for more data from an ApplicationMediator. The fireViewEvent method and the ViewEvent listeners are used to process the event (step 3116). If another ViewController needs
20 to be displayed, use a ViewEvent to represent that request and have the ApplicationMediator process that request.

VIII. Create ValidationRule(s)

Turning now to Figure 32, a flowchart of a process
25 for creating ValidationRules is depicted in accordance with a preferred embodiment of the present invention. A ValidationRule implements two methods: edit and normalize. A particular business validation rule is selected (step 3200) and a class that extends
30 ValidationRule is created (step 3202). This class should have a suffix of Rule to distinguish it as a

ValidationRule class. The edit() method is overridden (step 3204). The edit() method takes a user-inputted string and generates a formatted output for display. The normalize() method takes a user-inputted or formatted string and generates a normalized output for transmitting to some persistent storage (step 3206). If the normalize and edit methods have the same implementation, then implement in the edit() method and have the normalize() method call the edit() method. The ValidationRule will compare user input against the selected business rule (step 3208). If the user input is not valid, then a ValidationRuleException is thrown (step 3210) and the process terminates thereafter. If the user input is valid, the process also terminates.

15 C. Create a ViewEvent

With reference now to Figure 33, a flowchart of a process for creating a ViewEvent is depicted in accordance with a preferred embodiment of the present invention. In response to AWTEvents, ViewControllers visually manipulate components, containers and beans. A ViewEvent is created for an AWTEvent from a ViewController that needs action to be taken beyond the normal visual manipulation capabilities of the ViewController. ViewEvents are created by ViewControllers and processed by ApplicationMediators which are ViewListeners. Other ViewListeners can listen for ViewEvents to monitor and debug. The details are in two choices: 1) an interface is created to contain all the various ViewEvent codes used in the application (step 3300), 2) MyViewEvent is a subclass ViewEvent and contains the ViewEvent codes. Create an instance of

ViewEvent using one of the above codes (step 3302). Data can also be sent with the event and send the ViewEvent using fireViewEvent method of ViewControllerImpl (step 3304) with the process terminating thereafter.

5 D. Create an ApplicationMediator

 An ApplicationMediator class is typically present for every application function or use case of the system to be developed. An ApplicationMediator creates one or more ViewControllers to complete its function. It may
10 also create other ApplicationMediators for nested functions or use cases. With reference now to Figure 34, a flowchart of a process to create an ApplicationMediator is depicted in accordance with a preferred embodiment of the present invention. A particular function is selected
15 (step 3400) and create a class that extends ApplicationMediatorImpl (step 3402). This class should have a suffix of AM to distinguish it as an ApplicationMediator class. The init() method is overridden (step 3404). The ViewControllers used by this
20 ApplicationMediator are created (step 3406). The initViewControllers() and initApplicationMediators() methods can be used to create instances of the classes given a list of class name strings. These init methods also add the current ApplicationMediator as the
25 ViewListener for the newly created instances. The processViewEvent() method is overridden (step 3408).

 ViewEvents from all ViewControllers and ApplicationMediators created by the current ApplicationMediator are processed in this method. This
30 method is called in a separate thread from the AWTEvent thread by the viewEventPerformed() method of

ApplicationMediatorImpl, so that user processing can continue. RequestEvents are created for any requests for data or transaction to be processed by business logic outside of the client application (step 3410). These requests can be synchronous or asynchronous. For asynchronous RequestEvents, the requestResponse() and requestException() methods must be overridden. Refresh() is invoked on the ViewController or ApplicationMediator based on the response of the RequestEvent (step 3412) with the process terminating thereafter. Other possible actions based on ViewEvents include, for example, passing control to other ViewControllers or ApplicationMediators. Also, the ViewEvent can even be sent to higher level listeners using the fireViewEvent method of ApplicationMediatorImpl.

E. Create a RequestEvent

RequestEvents contain a major code to determine the Destination of the request and a minor code to determine the type of request. Turning now to Figure 35, a flowchart of a process for creating a RequestEvent is depicted in accordance with a preferred embodiment of the present invention.

An interface for containing the strings of major and minor codes of RequestEvents is created (step 3500). Major codes determine the Destination but should be based on a grouping of related request types. In this manner, the processing of these requests can be changed as a group. When processing a ViewEvent in an ApplicationMediator, a RequestEvent can be created for requesting data or a transaction from a business logic subsystem that may be outside of the client application.

An instance of RequestEvent is created using the appropriate major and minor code (step 3502). The event is sent using fireRequestEvent method of ApplicationMediatorImpl (step 3504) with the process
5 terminating thereafter.

F. Create a Destination

A Destination takes a RequestEvent and sends it to the appropriate location for processing including servers, the hosting client environment, an in memory
10 algorithm, or local devices. Each major code can be mapped to a different Destination. This allows the client application to access multiple servers and change locations of server processing without requiring code changes in the client application.

15 With reference now to Figure 36, a flowchart of a process for creating a Destination is depicted in accordance with a preferred embodiment of the present invention. A particular Destination for processing client requests is selected (step 3600) and a class that
20 extends DestinationImpl is created (step 3602). In the depicted examples, this class has a suffix of Destination to distinguish it as a Destination class. The init() method is overridden (step 3604). Any initialization required to connect to a particular Destination can be
25 implemented in this method. The requestEventPerformed() method is overridden (step 3606). A RequestEvent is passed to this method. The method should send the request to the appropriate Destination and update the RequestEvent with the response. Use the setStatus
30 method of RequestEvent to update the processing status of the request. If a timeout value has been set, then

implement a timeout mechanism for processing the request.

The Destination should observe the RequestEvent consume variable as an indicator that processing should be canceled. Next, the finalize() method is overridden
5 (step 3610) with the process terminating thereafter. If connection to a Destination needs to be cleaned up, that functionality should be located in this method. The TopListener or any other class that starts the application should create Destinations and register them
10 with the Transporter based on the major codes of the RequestEvents the Destination handles.

G. Create a TopListener

A TopListener manages the overall application. The
15 TopListener is the entry point into the hosting client desktop environment, which includes the operating sytem, a browser, and legacy programs. Typically the class that contains the top frame of the application will also implement the TopListener interface. This class should
20 have a suffix of Top to distinguish it as a TopListener class.

With reference now to Figure 37, a flowchart of a process for creating a TopListener is depicted in accordance with a preferred embodiment of the present
25 invention. A class is created that contains or is the top frame of the application and implements the TopListener interface (step 3700). An exit() method is created (step 3702) and a launch() method is created (step 3704). Next, a message() method is created (step
30 3706) and a title() method is created (step 3708). A TopEvent is created and sent to the TopListener with the fireTopEvent method. A major code of TopEvent.EXIT

indicates that the ApplicationMediator is finished. A major code of TopEvent.LAUNCH indicates that the ApplicationMediator needs an operating system program to be invoked. A major code of TopEvent.MESSAGE indicates that the ApplicationMediator needs a desktop specific message to be displayed. A major code of TopEvent.TITLE indicates that the ApplicationMediator needs a desktop specific application title to be displayed.

H. Create a PlacementListener

10 A PlacementListener is used to place an actual component, container or bean (i.e. ViewController.getComponent()) on a frame or window. With reference now to Figure 38, a flowchart of a process for creating a PlacementListener is depicted in accordance with a preferred embodiment of the present invention. A single type of placement is selected (step 3800) and creates a class that implements the PlacementListener interface (step 3802). This class should have a suffix of Placement to distinguish it as a PlacementListener class.

20 The placementEventPerformed method is then implemented (step 3804) with the process terminating thereafter. An ApplicationMediator and a Java Component is passed in to the method. The component should be added to some container managed by the PlacementListener class.

I. Assembling the JTC Program

The JTC pattern for building most client applications is now given. This description is the JTC pattern. A Destination "D" object is created and

Docket No. A-99-339

initialized. A Transporter object "T" is created and initialized. A major code "MC" is defined. "D" is added to "T" as a DestinationListener using major code "mc".

A main program "P" that implements the JTC,
 5 PlacementListner and TopListener interfaces is created and it creates top level frames.

An ApplicationMediator "AM" object is created. Transporter "T" is added as a RequestListener. Main program "P" is added as a PlacementListener and a
 10 TopListener. ApplicationMediator "AM" is initialized. ApplicationMediator "AM" creates and initializes one or more ViewControllers "VC" and immediately fires a PlacementEvent to start the program visuals.

This is the Java code for the JTC pattern:

```

15 public class MyProgram implements JTC,
                                   PlacementListener, TopListener {

    public static void main (String[] ) {

20         MyProgram P = new MyProgram();
           P.createFrames();

           MyDestination D = new MyDestination();
           dest.init();

25         MyTransporter T = new MyTransporter();
           trans.init();

           String MC = "mc";

30         T.addDestinationListener(MC, D);

           MyApplicationMediator AM =
  
```



```
new MyApplicationMediator();
```

```
AM.addPlacementListener(P);
```

```
AM.addTopListener(P);
```

```
5 AM.addRequestListener(T);
```

```
AM.init();
```

```
}
```

```
}
```

VI. Subsystem Behavior

10 A. ViewController Subsystem Runtime Behavior

With reference now to Figure 39, a diagram illustrating runtime behavior of a ViewController subsystem is depicted in accordance with a preferred embodiment of the present invention. The runtime behavior of the ViewController subsystem 3900 is shown in Figure 39. This subsystem is the basic GUI building block and the primary mechanism for interacting with the user. The ViewController interface 3902 extends JTC interface 3904. It defines the graphic components of an application and specifies methods for managing ViewListeners 3906, permissions, properties, resources, visibility and validity. It also specifies methods to refresh data. The ViewController is also responsible for handling refresh calls 3908 and updating the GUI accordingly.

ViewController 3902 is responsible for communication of information to ViewListeners 3906. ViewController 3902 knows the full semantics of the data model. One communication technique for sending information to ViewListeners 3906 is to fire a ViewEvent 3910 that the

listeners can interpret. The specification of the major and minor codes in ViewEvent 3910 directs the listeners as to what action to perform. Additional information in the form of a String or user-defined data object can be specified by the setData method. ViewController 3902 also can provide named methods. Once a ViewEvent 3910 is consumed, it is not forwarded to other listeners. Another communications technique is to use property change events on the data model. Management of data and use of this approach is discussed explicitly in a later section.

The actual implementation of ViewController interface 3902 is performed in the ViewControllerImpl class 3912. A ViewControllerImpl creates a panel 3914 that allows interaction with the user. All AWTEvents 3916 are sent to a user-defined subclass 3918, which extends the ViewControllerImpl 3912. Java provides two built-in mechanisms for managing panels: the Panel class 3920 that is part of the Java AWT and the Jpanel class 3922 that is part of the com.sun.java.swing package. ViewControllerBaseImpl 3924 is a superclass of ViewControllerImpl 3912, which abstracts these different panel mechanisms.

ViewController subsystem 3900 is also responsible for validating data formats. ValidationRules 3926 themselves are not part of the ViewController 3902, but the ViewController 3902 performs this validation between the input and output using the standard AWT methods such as getText and setText. The edit method inputs a transmittable string and outputs it in visible form (e.g., 1234 to \$12.34) while the normalize method inputs a string in visible form and converts it to

transmittable form (e.g., \$12.34 to 1234). A failure in format conversion results in a ValidationRuleException 3928.

For instance, data starts out in a text field (component) in a ViewController (component) that derives from JPanel (container). The value of the text field is retrieved and passed as a String to the validation rule. The result will be a validated and formatted string or an exception. In the former case, the textfield is redisplayed with the new contents AND/OR the data model is updated. In the latter case, some error message is displayed.

With chaining, the results of each validation rule call are passed to the next validation rule. Processing will stop when the first exception is encountered (first exception first return (FEFR)) or all validation rules have been processed.

Turning next to Figure 40, steps in the operation of a ViewController subsystem, as viewed from a ViewControllerImpl, are depicted in accordance with a preferred embodiment of the present invention. The process begins by implementing the ViewController interface (step 4000). Thereafter, the JTC interface is implemented (step 4002). Thereafter, specific methods are added for the ViewController (step 4004). Thereafter, the GUI is created and composed (step 4006). Thereafter, a reference to "yourself" is returned in getComponent (step 4008). ViewControllerBaseImpl or any subclass of ViewControllerImpl can return the particular component to be placed (by the PlacementListener) in the getComponent() method. Thereafter, permission keys, resources, and properties are returned when requested

(step 4010).

The permission keys, resources, and properties are updated when requested (step 4012). Internal AWTEvents are handled (step 4014). Data fields are validated and formatted (step 4016). ViewEvents are issued for semantic interpretation (step 4018). The AWT thread is returned (step 4020). Refresh calls are handled to update the GUI (step 4022). Data is accessed (step 4024). A determination is made as to whether the ViewController is still active (step 4026). If the ViewController is still active, the process returns to step 4010. Otherwise, the process terminates for the ViewControllerImpl.

Additionally, a ViewController may be used to generate alternate views. For example, the application containing the view controller may be located on the server in which a placement listener is used to call a method in the ViewController to generate a presentation of a component in an alternate format, for example to be displayed in an HTML browser or an XML browser. In this example, the ViewController is on the server while the alternate view is displayed on the client. The PlacementListener has one method: placementEventPerformed that takes a PlacementEvent as a parameter. The PlacementEvent contains a major, minor, source ApplicationMediator and Component (from getComponent() of the ViewController). To use the alternate view, and without extending the PlacementEvent definition, it should be noted that to call for the alternate view, the ViewController needs to be accessed. Therefore, the ViewController can be accessed via the Object data; variable in the PlacementEvent.

Alternate views, may include, for example, text for text based terminals or event data for a debug monitoring display. also, any user input from HTML or other view comes back to ApplicationMediators and ViewControllers as refresh data. The ApplicationMediator may either display the next ViewController or have current ViewController redisplay the entered user input.

With reference now to Figure 41, a flowchart of an alternate view process is depicted in accordance with a preferred embodiment of the present invention. The processes of the present invention may be applied to other types of applications other than clients. For example, the process of the present invention may be applied to create applications for use on a server. In Figure 41, the process of the present invention is used to generate an alternate view in which the containers are output in the form of HTML for transfer to a client over a communications link.

In Figure 41, the application mediator receives a ViewEvent from a ViewController (step 4100). A determination is made as to whether the ViewController should be placed in the client (step 4102). If the ViewController should not be placed, then the process terminates. If the ViewController should be placed, then a PlacementEvent is created with the proper major code (step 4104). This PlacementEvent is then fired (step 4106). A PlacementListener is invoked (step 4108). The PlacementListener receives the PlacementEvent (step 4110). A determination is made as to whether the major code is for a placement HTML (step 4112). If the major code is for an HTML placement then a call is made by the PlacementListener to toHTML method in the ViewController.

(step 4114). In response, the view components are translated into HTML components (step 4116). An HTML output is generated (step 4118). This HTML output is then sent to the client browser for display (step 4120) with the process then terminating.

With reference again to step 4112, if the major code does not indicate the use of HTML as an alternate view, then a determination is made as to whether the major code identifies another alternate view (step 4122). If another alternate view is not indicated by the major code, then the process terminates. Otherwise, an alternate view is created and placed (step 4124) with the process then terminating. The other alternate view may take various forms, such as, for example, XML. Step 4122 may be iterated for a number of different alternate views that may be provided through the ViewController component. In this manner, the architectural pattern of the present invention may be applied for use on a server. This mechanism is used by a PlacementListener to call a method in a ViewController to create an HTML version of an existing screen. The mechanism for creating the HTML view is application dependent/screen dependent in the depicted examples.

An example of how to configure JTC with alternate views on a server is given. The deployment of JTC on the server is under the control of a servlet. Since, as described in this example, HTML is in the client, when a client browser invokes a URL submit, the web server obtains the request and passed control to a servlet. The servlet obtains a key/value pair list of values entered in the HTML client. This list is passed to the ViewController alternate view being displayed. The

ViewController iterates over the list and applies its ValidationRules. If a ValidationRuleException is generated, the servlet indicates an error message to the HTML client. If a successful result is obtained, the
5 ViewController will generate a ViewEvent, as mentioned above, causing the alternate view cycle through the ApplicationMediator and PlacementListener to repeat. The primary benefit being that one source code system can be maintained and versioned to produce multiple client
10 channel views.

Turning now to Figures 42 and 43, diagrams detailing processes within the ViewController subsystem are depicted in accordance with a preferred embodiment of the present invention. The code illustrated is Java code in
15 the depicted examples.

The creation of a ViewEvent based on an ActionEvent from the Java AWT is shown in Figure 42. The actionPerformed method detects the type of action and constructs a corresponding ViewEvent. The source of the
20 ActionEvent was a click of the nextButton. This is detected and the ViewEvent ve is created with the self-referential "this" pointer being passed in. The major field is set to NEXT, a predefined type, and the event is fired. ActionPerformed has done its job and
25 returns.

Figure 43 shows how a ViewEvent is processed by a . Before any events can be processed, an object has to add itself as a listener for the appropriate ViewEvents. In this case, the listener is added to the
30 customerDetailsViewController. The method processViewEvent may carry out some initial action for all events and then a switch statement is used to

separate out all of the major code fields of interest. The comments inside the switch indicate where different actions can be performed for different events. A complete list of predefined major event codes is given in

5 **Figure 44.**

Validation rules either use edit to convert from a transmittable string to the viewable format or use normalize to convert from a viewable format to a transmittable string. Figure 44 shows how a text field representing a social security number can be put into visual form (e.g., 123-45-6789) by calling SocialSecurity.edit. Since a ValidationRuleException will be thrown if the conversion is unsuccessful, this call is embedded in a try statement with a catch for exception handling. In a comparable manner, Figure 46 shows how the visual format for a social security number can be converted back to a transmittable format.

It is possible to build a chain of validation rules.

Figure 47 illustrates the application of two edit rules, range checking (that is, the value is within specific limits) and formatting for viewing. The separate rules are assigned to string values named range and money. This string values are then put into an array of strings. The transmittable form of the data is stored in value. The method applyEdits applies both validation rules to this data; if any exception occurs, it is handled by the catch portion of the try statement. The setText method from the Java AWT is used to redisplay the formatted data.

30 Two techniques exist for creating the ViewControllerBaseImpl class: by inheritance and by delegation. Figure 48 illustrates inheritance where the

ViewControllerBaseImpl is a subclass of JPanel from the Java com.sun.java.swing package. Notice that the operation getComponent simply returns a pointer to itself. When delegation is used, as shown in Figure 49, 5 ViewControllerBaseImpl implements the ViewController interface. The constructor for a user defined XYZ class creates a new instance, called xyz. Operations from the ViewControllerBaseImpl class are translated into corresponding operations from the XYZ class. Three 10 operations are illustrated: getComponent, setEnabled, and setVisible.

B. ApplicationMediator Subsystem Runtime Behavior

Turning now to Figure 50, a diagram illustrating runtime behavior of an ApplicationMediator subsystem is 15 depicted in accordance with a preferred embodiment of the present invention. The runtime behavior of the ApplicationMediator subsystem 5000 is shown in Figure 50. ApplicationMediator 5002 manages all the major components of the thin client architecture JTC 5004. This includes 20 managing instances of ViewControllers 5006 and ApplicationMediators 5008, and adding, firing, or removing instances of PlacementListeners 5010 and TopListeners 5012. The implementation of the interface of ApplicationMediator 5002 is carried out by the class 25 ApplicationMediatorImpl 5014. An application dependent subclass 5016 of ApplicationMediatorImpl 5014 actually processes the various events in the subsystem 5000. ApplicationMediator subsystem 5000 can process ViewEvents 5018 generated by ViewController 5020 and it can generate 30 its own ViewEvents if necessary. Requests for actions to be performed by the server result in RequestEvents 5022.

A subclass is needed to implement a state machine for correctly ordering operations. Sample operations include which ViewControllers 5006 to allocate or which to make visible, when to fire or re-fire a ViewEvent, when to fire a RequestEvent 5022, and when to fire a TopEvent 5012.

The typical sequence of actions when a ViewEvent is received is to request a PlacementListener 5010 to place another ViewController 5006 on the screen, to request TopListener 5012 to do something with the operating system desktop, and to fire RequestEvents 5022 for fulfillment by a server. Refresh calls are propagated to the ViewControllers 5006 and ApplicationMediators 5008. All JTC objects allocated are returned via the getJTCs method.

Turning next to Figure 51, a diagram illustrating Event threading support is depicted in accordance with a preferred embodiment of the present invention. Figure 51 illustrates the two styles of AWT Event threading support. This occurs when a ViewController 5100 generates a ViewEvent 5102 that is to be processed by the ApplicationMediator 5104. Style 1 uses a queuing mechanism while style 2, the default, uses thread dispatching.

The following steps occur when using the queuing approach: create a sleeping thread during initialization, when a ViewEvent arrives place it in a queue 5106 of ViewEvents, notify the sleeping thread and return to the calling ViewEvent thread (typically an AWT Event thread). The run() method 5108 wakes up to process the ViewEvent by taking it off the queue. The processViewEvent method is

then called. Subclasses override this method to handle the ViewEvent directly.

The second mechanism is to use thread dispatching directly. When a ViewEvent arrives, create and start a
5 thread 5110 passing it the ViewEvent and return to the calling ViewEvent thread. Add the new thread to a list to provide a handle to access it. This new thread invokes the processViewEvent method which subclasses can override to handle the ViewEvent directly.

10 With reference now to Figure 52, a flowchart of a process used in designing and executing an ApplicationMediator is depicted in accordance with a preferred embodiment of the present invention. The process begins by implementing an ApplicationMediator
15 interface (step 5200). Thereafter, the JTC interface is implemented (step 5202). ApplicationMediatorImpl methods are added (step 5204). ViewControllers are created using the initViewController method (step 5206).

Thereafter, initApplicationMediator are used to
20 create other ApplicationMediators as necessary (step 5208). Thereafter, the process listens for ViewEvents and RequestEvents (step 5210). A determination is made as to whether a ViewEvent has been received (step 5212). If a ViewEvent has not been received, the process returns to
25 step 5210. Otherwise, a request is made to a PlacementListener to put another ViewController on the screen (step 5214). This step basically results in the display of another ViewController's getComponent() on the screen.

30 Next, the TopListener performs desktop operations as appropriate (step 5216). The ApplicationMediator also fires the appropriate RequestEvent to server (step 5218).

Refresh calls are sent to ViewControllers and ApplicationMediators (step 5220). All JTC objects allocated are returned (step 5222) with the process then returning to step 5210.

5 C. Placement Subsystem Runtime Behavior

With reference now to Figure 53, a diagram illustrating runtime behavior of the Placement subsystem is depicted in accordance with a preferred embodiment of the present invention. The runtime behavior of the Placement subsystem is 5300 shown in Figure 53. The JTC architecture separates out the actions of ViewEvent 5302 creation (done by ViewController, 5304), view ordering (done by ApplicationMediator, 5306), and view placement (done by PlacementListener, 5308). A PlacementEvent 5310 is used to notify a PlacementListener that a ViewController needs to be adjusted on the screen.

The state of a PlacementEvent includes a major code, which is the primary identifier of the PlacementEvent. A minor code serves as the secondary identifier of the PlacementEvent. The PlacementEvent state also includes a source identifying the source object that creates the Event. A java.awt.Component also is present in the state that identifies the component to be placed. The state also contains consume, which identifies whether an PlacementEvent has been consumed. A consumed PlacementEvent will cause the JTC to stop processing the PlacementEvent immediately. Additionally, the PlacementEvent state also may contain other data, which may be, for example, a reference to generic object data that specifies placement information specific to the application.

Each PlacementEvent 5310 contains a placement type (ADD, ADDALL, REMOVE, REMOVEALL, MODIFY, SHOW, DONE, FREEZE , and REPAINT are predefined), a minor code for additional information, the sender of the event, the component, and provisions for additional generic data. Typically an ApplicationMediator 5306 will assign the appropriate major code, find the component using the getComponent method, create PlacementEvent 5310 making itself the sender, and fire the placement event. The PlacementListener 5308 uses placementEventPerformed to process PlacementEvent 5310 based on the placement type and the source of the event.

Details of the steps discussed above are now examined through various examples of Java code. The creation of a PlacementEvent from an ApplicationMediator is shown in Figure 54. There are three parameters to a PlacementEvent constructor: the object (referenced by "this") creating the event, the component involved, and the major code. In this example, the major code is set to ADD and the component comes from customerDetailsViewController. After the PlacementEvent is created, it is fired by the ApplicationMediator.

The PlacementEvent is processed by a PlacementListener, as shown in Figure 55. The method placementEventPerformed decides on the appropriate action based on the major code and the source. In this example, the major code is ADD, which is one of the cases in a switch statement. If the source is PreferencesAM, then the PlacementEvent component is centered in panell, otherwise it is added to panel2.

Turning now to Figure 56, a flowchart of a process used in processing a PlacementEvent is depicted in

accordance with a preferred embodiment of the present invention. The process begins by one or more ViewControllers responding to GetComponent (step 5600). After, ApplicationMediators control the ordering of the
5 ViewController (step 5602). The ApplicationMediators fire PlacementEvents (step 5604). The PlacementListeners select the proper implementation and place the Component on the screen (step 5606) with the process terminating thereafter.

10 D. TopListener Runtime Behavior

With reference now to Figure 57, a diagram illustrating runtime behavior for a TopListener subsystem is depicted in accordance with a preferred embodiment of the present invention. The runtime behavior of the
15 TopListener subsystem 5700 is shown in Figure 57. TopListener 5702 performs generic operations on the business desktop, as specified by a TopEvent 5704. The TopEvent and TopListener have several functions. One function is interaction with the host environment (i.e.
20 for a Java application--the operating system), with the host environment services (i.e. other applications on the operating system), with the application enabler (i.e. the Netscape browser and its status line or maybe aspects of the Java Virtual Machine), and with host environment
25 policies (i.e. how is shutdown handled, how are error messages handled). The hosting client desktop environment is an embodiment of one or more of these items.

The TopListener is also used by the JTC application
30 to send events to itself. For example, based on some heuristic, the TopListener may decide to change the

Destination from one using sockets to one using URLs. Or, a second destination may be added to perform logging. The permissions also may be altered. The TopListener may use a heuristic to decide to invoke hookJTCs and hookAWTs on the application and start tracing. Thus, in addition to accessing hosting services, the TopListener also allows self alterable JTC actions to occur on the JTC application.

TopEvent 5704 contains a major code, a minor code, the source that created the event, a consumed event that causes the JTC to stop processing it as soon as possible, and a generic data object. There are predefined constants for EXIT, BROWSER, TITLE, STATUS, OS, TRACE, DEBUG, LOG, HOOKAWT, and HOOKJTC. ApplicationMediator 5706 uses addTopListener to pass itself to the TopListener for later callback in the topEventPerformed method.

Details of the steps discussed above are now examined through various examples of Java code. With reference now to Figures 58-60, Java code used in a TopListener is depicted in accordance with a preferred embodiment of the present invention. The creation of an ApplicationMediator and the adding of a PlacementListener is shown in Figure 58. The creation and firing of a TopEvent in an ApplicationMediator is shown in Figure 59. Java code illustrating the callback to a TopListener with a TopEvent and inspecting the TopEvent for semantic interpretation is shown in Figure 60. This method uses the major code to separate out the various cases and to perform the appropriate actions.

E. RequestEvent Subsystem Runtime Behavior

The RequestEvent subsystem provides a mechanism for sending messages between various components in the architectural pattern of the present invention. This subsystem provides a mechanism to access data and to send data between components, such as an ApplicationMediator, a Transporter, and a Destination. With reference now to Figure 61, a diagram illustrating runtime behavior of a RequestEvent subsystem is depicted in accordance with a preferred embodiment of the present invention. The runtime behavior of the RequestEvent subsystem 6100 is shown in Figure 61. An ApplicationMediator 6102 receives a ViewEvent 6104 from a ViewController 6106 and determines it has to issue a RequestEvent 6108 to an appropriate service provider. RequestEvent 6108 can either be synchronous or asynchronous in the depicted examples. RequestEvent 6108 has fields for a major code (a request family), a minor code (a specific request), a user-defined version, a status string showing the current stage of processing, a possible consume event that will cause the JTC application to stop processing it as soon as possible, and a data field for a generic data object.

ApplicationMediator 6102 calls fireRequestEvent. A Transporter, which is one example of a RequestListener 6110, is responsible for forwarding all RequestEvents to one or more destinations based on the major code. ApplicationMediator 6102 will be called back in one of two ways: requestResponse to indicate success and passed a RequestEvent that may have new data or requestException to indicate failure and passed a RequestException with the reason for failure.

With reference now to Figure 62, Java code illustrating the creation of RequestEvent, setting its major and minor codes, and firing an asynchronous RequestEvent from an ApplicationMediator is depicted in accordance with a preferred embodiment of the present invention. Details of the steps discussed above are now examined through various examples of Java code.

The creation, firing, and callback of a RequestEvent in an ApplicationMediator is shown in Figure 62. After the RequestEvent *r* is created, the major and minor codes are set to "Loans" and "SubmitCustomerInfo" in this example. The event *r* is fired asynchronously by the ApplicationMediator. This is done inside of a try statement so that any RequestException can be caught and processed appropriately in Figure 63. Later there will be a callback to the ApplicationMediator by requestResponse to indicate successful processing of the RequestEvent or by requestException to indicate failure processing the RequestEvent in Figure 64.

With reference now to Figure 65, a flowchart of a process for using a RequestEvent is depicted in accordance with a preferred embodiment of the present invention. The process begins by an ApplicationMediator receiving a ViewEvent from a ViewController (step 6500). Thereafter, the ApplicationMediator creates a RequestEvent specifying the major code, and a version (step 6502). Thereafter, a fireRequestEvent is issued to cause the RequestEvent to be sent to a service provider (step 6504). Based on the major code, the Transporter forwards the RequestEvent to one or more Destinations (step 6506) with the process terminating thereafter.

F. Transporter Subsystem Runtime Behavior

The Transporter subsystem is used to map and send various RequestEvents to different Destinations . With reference now to Figure 66, a diagram illustrating runtime behavior of a Transporter subsystem is depicted in accordance with a preferred embodiment of the present invention. The runtime behavior of the Transporter subsystem 6600 is shown in Figure 66. The main task of the Transporter class 6602 is to map RequestEvents 6604 received from an ApplicationMediator 6606 to the appropriate Destinations. A Destination 6608 is added to the Transporter 6602 by using addDestinationListener. Each listener is associated with a destination major code. In the depicted examples, multiple destinations can listen for the same major code. Events are processed in a First-In First-Out, First Exception First Return (FIFO/FEFR) 6610 fashion in these examples. It is possible to have one Destination pass a message to another Destination. RequestExceptions 6612 are used to handle failures.

Turning now to Figure 67, Java code illustrating creation of a Transporter and adding it as a RequestListener is depicted in accordance with a preferred embodiment of the present invention. The Transporter t and ApplicationMediator am are both created. The method addRequestListener is called to add t as the transporter for am.

G. Destination Subsystem Runtime Behavior

A Destination subsystem provides a number of different functions, such as, for example, locating a server, creating low level protocols, such as sockets,

URLs, RMI, and files for network transport. Additionally, the Destination subsystem provides a mechanism to communicate with different servers, such as EJB, servlets, web servers, application servers, and other legacy systems. Additionally, message formats are provided to turn RequestEvents and data into formats for the target or server. Additionally, the Destination subsystem will finish the job in an appropriate amount of time or cancel the operation in response to a timeout. Additionally, the Destination subsystem also determines how many times a connection should be retried. Additionally, the Destination subsystem caches data until it becomes stale. Additionally, the Destination creates all data model objects used in the JTC client application. Pre-processed and post-processed RequestEvents are also logged along with locating and creating objects for processing requests.

With reference now to Figure 68, runtime behavior of a Destination subsystem is shown in accordance with a preferred embodiment of the present invention. In Destination subsystem 6800, Destination class 6802 and its implementation, DestinationImpl 6804, handles all the details to locate server 6806, use various transport types and server types 6808, and, in general, keeps track of where objects are. Destination codes are either named, given a wildcard value (*, means interested in all requests) or given a priority value (!, means interested in all requests and has priority). Some special cases of priority are FIRST, SECOND, ..., NINTH. DON'T_CARE is a special case of wildcard that indicates even if a RequestException is present, do not stop processing and send the request to the Destination.

Details of the steps discussed above are now examined through a Java code example. In Figure 69, a diagram of Java code for creation of a Destination and adding it as a destination listener is depicted. Turning now to Figure 5 69, Java code illustrating creation of a Transporter and adding it as a RequestListener is depicted in accordance with a preferred embodiment of the present invention. The Destination d and the Transporter t are both created. The method addDestinationListener is called to add d as a 10 destination listener for transporter t using the major code "Loans."

Turning next to Figure 70, a diagram illustrating Java code for creating Destinations with wild card, priority and normal major codes, firing a RequestEvent, 15 and a report of the expected results is depicted in accordance with a preferred embodiment of the present invention. Transporter t and WildDestination d are created and d is added as a WILDCARD destination listener. Destination d is now assigned to be a EJBDestination and 20 added to t with the major code of "Loans." Destination d is then assigned to be a PriorityDestination and added to t with the major code of PRIORITY. Finally d is assigned to be a LoggerDestination and added to t with the major code of DON'T_CARE. At a later time, the RequestEvent r 25 is created with the major field of "Loans" and is fired. Event r will be sent to PriorityDestination 1st, EJBDestination 2nd, WildDestination 3rd, and LoggerDestination last. Event r reaches the first three destinations only if no RequestExceptions are thrown; it 30 reaches LoggerDestination regardless of any RequestExceptions.

VII. Component Details and Applications

A. use of getJTCs

The architectural pattern of the present invention as
5 all major objects in a JTC application implement the JTC
interface. This includes the driver programs that
launches the Application or applet. This interface allows
referencing of all objects with a consistent Java type.
The behavior includes, clear, exit, getJTCs, init,
10 isEnabled, setEnabled, and toString. Interfaces extending
JTC are ViewController, ApplicationMediator, and
Destination. ViewControllerImpl, ApplicationMediatorImpl,
Transporter, and DestinationImpl are classes implementing
JTC. Consider a program called "program" that implements
15 all of "significant" the Java and user defined interfaces.
A significant interface is an interface in which the
"program" is interested in receiving callbacks. The
getJTCs method causes each JTC object to return a list of
other JTC objects that the JTC object creates. This
20 method may be used by the "program" to iterate over this
list and inspect each object's types and add the
"program" as a listener to all of the object's significant
event listener types.

As a result, the system may be reconfigured through
25 this mechanism. For example, the Transporter may be
disabled or a priority Destination may be added using
this method. Further, this method provides a mechanism
for non-intrusive logging, tracing, debugging, and
monitoring. This is particularly significant when a
30 ViewController is encountered. In addition to adding the
"program" as a ViewListener, the getComponent method is
called to retrieve a java.awt.Component. The

java.awt.Component object is at the top of the Java graphics Component hierarchy. The "program" recursively looks at this hierarchy, traverses it, and add the "program" as listeners for all significant Java AWTEvent and JFC events. Using reflection, any Java object that is returned can be inspected and "hooked" for significant interface events. The "program" can do various tasks with these events such as trace, remotely display the events, reroute the client application to a new server, and record user statistics; all non-intrusive to the application in terms of writing additional code to support this capability.

With reference now to **Figure 71**, a diagram of Java code used for accessing, identifying type and recursively attaching JTC, AWT and JFC listeners to processing accessible JTC, AWT and JFC programs and objects JTCs is depicted in accordance with a preferred embodiment of the present invention. In particular, all JTCs are found and put into a Java Vector. Additional actions are performed depending on the type of JTC, such as adding RequestListeners for ApplicationMediators or adding Viewlisteners for ViewControllers. This function continues processing all JTC objects in a recursive manner.

The function is named hookJTCs and it starts out at the root JTC. The Vector jtcS is initialized to null. The method getJTCs is used to find all JTCs of the root. If an exception occurs or no JTCs are found, the function is exited.

Assuming the vector is not empty, each JTC object in the vector is processed in turn. This code illustrates the "program" adding itself as a RequestListener and

ViewListener to ApplicationMediators. This code illustrates the "program" adding itself as a ViewListener to ViewControllers. Although not shown in the code, other JTC objects, such as a Transporter, might be hooked by the "program". The function hookJTCs is called recursively at this point resulting in the JTC hierarchy being processed in a depth-first then breadth-first fashion.

With reference now to Figure 72, an example of a test program is depicted in accordance with the preferred embodiments of the present invention. In Figure 71, the main program Test1.java stores a static reference to itself so that a launcher can call getJTCs() and hook the application. Figure 73 is a diagram of Java code used for attaching AWT and JFC listeners to AWT and JFC containers, components and beans depicted in accordance with a preferred embodiment of the present invention. Figure 74 is a diagram of Java code used for attaching AWT and JFC listeners to AWT and JFC components (java.awt.Button, com.sun.swing.java.JButton and com. and com.sun.java.swing.JTextField) depicted in accordance with a preferred embodiment of the present invention.

With reference now to Figure 75, a flowchart of a process for accessing objects is depicted in accordance with a preferred embodiment of the present invention. The process begins by receiving a call to get JTCs and get the vector of JTC object (step 7500). A determination is then made as to whether a return is null (step 7502). If the return is null, the process then returns. This indicates that no objects are present. Otherwise, a determination is made as to whether more entries are present (step 7504). If more entries are present, then an unprocessed entry is used (step 7506). Thereafter, a determination is

made as to whether the entry is an instance of an ApplicationMediator (step 7508). If the instance is that of an ApplicationMediator, the "program" is added as a ViewListener (step 7510).

- 5 The process then adds the "program" as a RequestListener (step 7512) with the process then making a recursive call and starting at step 7500.

- With reference again to step 7508, if the entry is not an instance of an ApplicationMediator, a
10 determination is made as to whether the instance is that of a ViewController (step 7514). If the object for the entry is an instance of a ViewController, the "program" is added as a ViewListener. Thereafter, a hookAWTs with the component returned from the ViewController's
15 getComponent method is performed (step 7518). This step is described in more detail below with respect to Figure 77. The process then makes a recursive call and starts at step 7500.

- With reference again to step 7514, if the instance
20 is not a ViewController, then a determination is made as to whether the object for the entry is an instance of a Transporter (step 7520). If the object is an instance of a Transporter, the "program" is added as a RequestListener (step 7522) with the process then making
25 a recursive call and starting at step 7500.

- With reference again to step 7520, if the object is not an instance of a Transporter, a determination is made as to whether the object is an instance of a component (step 7524). If the object is an instance of the
30 component, the process performs hookAWTs with the component (step 7526) with the process then proceeding to

step 7504. Step 7526 is used to hook AWTs to a component. With reference again to step 7524; if the object is not an instance of a component, the process then returns to step 7506 to select another entry for processing.

- 5 With reference again to step 7504, if more entries are not present, then a non-intrusive program, the "program", is run to listen for events and debug, log, trace, and/or simulate (step 7528). The debug, log, trace, and/or simulate functions may be performed using
- 10 presently available techniques for debugging, logging, tracing, and/or simulating applications.

B. Recursive Processing of a Component Hierarchy

- A component hierarchy is a recursive data structure
- 15 in the sense that, if the component is a container, then it may contain other components, which in turn may contain other components, and so forth. Consider the hierarchy shown in Figure 76. Component A 7600 is a container that is a list of three other components: A1
- 20 7602, A2 7604, and A3 7606. A1 7602 is a container with two components A11 7608 and A12 7610), and A3 7606 is a container with two components (A31 7612 and A32 7614). The components A11 7608, A12 7610, A2 7604, A31 7612, and A32 7614 are not containers. The goal of a "hook"
- 25 algorithm is to find all the components in the hierarchy and "hook" each "child" component to the "program" component in the sense of adding appropriate listeners for events, while passing through the "parent" container.

- Since the component hierarchy is a recursive data
- 30 structure, a recursive algorithm most easily processes this hierarchy. It starts at the root component, A BT00, and determines if it is a container. It proceeds to

process the first component in the container, A1 BT02, using the same algorithm. A1 7602 is also a container, so the algorithm is applied recursively to the component A11 7608. A11 is not a container, so a "hook" to this component (that is, appropriate listeners) is added. Control now returns to the algorithm processing A1 7602. There is another component in the list of A1 7602, namely A12 7610, and it is processed recursively. A12 7610 is not a container, so it adds listeners appropriate for its type. No more components exist in the list of components for A1 7602, so it has now added listeners for all its children. Control returns back to the algorithm processing parent A.

Component A proceeds to the next item in its container, namely A2 7604. A2 7604 is not a container, so it adds listeners appropriate for its type. A now has added all the appropriate listeners for its children, A1 7602 and A2 7604, and all of their descendants. A3 7606 is processed next and, in a manner analogous to A1 7602, it adds listeners for its two children, A31 7608 and A32 7614. Listeners have been added for all of the descendants of A 7600, so the algorithm terminates.

With reference now to Figure 77, a flowchart of a process for performing hookAWTs is depicted in accordance with the preferred embodiment of the present invention. This process is used to hookAWTs to a component. The processes depicted in Figure 77 are a more detailed description of steps 7518 and 7526 in Figure 75. The process begins by determining whether the component is a container (step 7700). If the component is a container; then a list of the components in the container is

obtained (step 7702). Thereafter, a determination is made as to whether more components are present in the list (step 7704). If more components are present in the list, then the next component is selected for processing (step 7706). A recursive call to D is performed (step 7708). This step results in the process returning to step 7700 and performing the steps as described above. Thereafter, the process hooks the component (step 7710). Then, the process returns to step 7704 to determine whether more components are present in the list.

If more components are not present in the list, then the component is hooked (step 7712) with the process and returns.

With reference again to step 7700, if the component is not a container, then the process also proceeds to step 7712. Step 7710 and step 7712 are described in more detail below in Figure 78.

With reference now to Figure 78, a process for hooking a component is depicted in accordance with the preferred embodiment of the present invention. Figure 78 is a more detailed description of the processes performed in step 7710 and 7712 in Figure 77. The process begins by determining whether the component is an instance of a button (step 7800). If the component is an instance of a button, the button is added as an action listener (step 7802) (with the process then returning). An Action Listener listens for events generated by Button, List, MenuItem, and TextField, all objects that can be part of the user interface. The event is handled by the method actionPerformed.

With reference again to step 7800, if the component

is not an instance of a button, a determination is made as to whether the component is an instance of a JButton (step 7804). Button is an object built using the AWT. JButton is an object built using the Swing library.

- 5 Swing components typically provide greater platform independence. If the component is an instance of a JButton, then the JButton is added as ActionListener, a ChangeListener, and an ItemListener (step 7806). A ChangeListener listens for events generated by JFC
- 10 components when their internal state has changed. The event is handled by the method stateChanged.

An ItemListener listens for events generated by Checkbox, CheckboxMenuItem, Choice, and List, all objects that can be part of the user interface. The event is

15 handled by the method itemStateChanged.

- With reference again to step 7804, if the component is not an instance of a JButton, a determination is made as to whether the component is an instance of a component (step 7808). If the component is an instance of a
- 20 component, then the component is added as an ActionListener and a CareListener (step 7810), with the process returning thereafter. A CaretListener listens for events generated by objects that contain text cursor positions (caret) and the caret changes in the position
- 25 of text. The event is handled by the caretUpdate method.

- With reference again to step 7808, if the component is not an instance of a component, a determination is made as to whether the component is an instance of other AWT/Swing/Bean types in Java (step 7812). If the
- 30 component is an instance of these other types, then the component is added to all event types generated by the component, with the process returning thereafter. If the

component is not an instance of these other types, the process then returns.

The getJTCs and getAWTs algorithms are easily extended to other object types and hierarchies by brute force coding of the "instanceof / addListener" tests or by using reflection on the objects to "search for interfaces, search for interface methods, addListener" as reflection supports.

10. C. Management of Data Objects

User interaction triggers ViewEvents to perform a variety of tasks. These include system tasks (e.g., open, close, cancel, etc.), status tasks (e.g., various messages), navigation tasks (e.g., next, previous, first, last, etc.), data tasks (e.g., add, delete, modify, etc.), assist tasks (e.g., search, find, help, etc.) and live (streaming) tasks (e.g. fast, medium, delay, timer, on, off, high, low, etc.).

Some tasks can be performed locally, but others are sent to a service fulfillment provider (the Destination) in the form of a RequestEvent containing appropriate data. Data is returned, often different than the data sent, and refresh is used to change what the user is viewing.

Details of the steps discussed above are now examined through various examples of Java code. With reference now to Figures 79-82, diagrams of Java code for use in managing data objects is depicted in accordance with a preferred embodiment of the present invention. Passing data for the main program to ApplicationMediators is shown in Figure 79. The ApplicationMediator aml and the data object named data are created. By using the refresh method, the main program passed this data into the

ApplicationMediator. Refresh accepts any data object regardless of the number or shape of objects. It is because the data type definition is "Object data;" that JTC data support is neutral to the actual classes and interfaces of the data models being used. It is because the refresh method uses a data type definition "Object data;" that additional data models can readily be integrated into a JTC client application with minimal effort and side effects. The ApplicationMediatorImpl passes the refresh on to all ApplicationMediators and ViewControllers.

Alternatively, if the results of a RequestEvent contains changed data, the ApplicationMediator can initiate the refresh on to all ApplicationMediators and ViewControllers.

The code in Figure 80 illustrates how a ViewController would handle data that uses the an example key-value pair data model. The first refresh function simply receives the data as type Object and type casts it to type KeyValue in a call to the worker refresh function of the key-value data model. The refresh of key-value pairs calls the AWT method setText for each of the textfields and then calls repaint() to change the user display (validate() followed by repaint() can also be used). The code in Figure 81 illustrates how this top level refresh method can input any object and route it to a more particular version of refresh by detecting the type of data and casting it to a more specific type. The types Vector, KeyValue, and XMLData are shown, but others can be added. Figure 82 illustrates the processing of data in the form of a Vector where the first element is of type Customer and the second element is the ID. These two

values are extracted from the vector, cast and then assigned to the appropriate fields by using setText. A repaint updates the screen.

Although the Destination locates, creates and packages these data types, the ViewController also knows the syntax of the data layout and packaging and the semantics of the data model.

ApplicationMediators and ViewControllers can also bypass the normal JTC ViewEvent notifications when data has changed. For example, refresh can use an implicit data model (i.e. String, Integer), a user defined data model (described above), named methods in the ViewController (less portable) and traditional Java property change event notifications. Further, live data models (i.e. streaming media, asynchronous device input) can also be incorporated easily. It is because JTC uses the data type "Object data;" and because the refresh method uses a data type definition "Object data;" that additional data models, event notifications, data access and data update differences can readily be integrated into a JTC client application with minimal effort and side effects - the GUI parts of the ViewController, ValidationRules, ApplicationMediators, ViewEvents, RequestEvents, TopEvents, PlacementEvents, TopListeners, and PlacementListeners do not have to change. Only small parts of the ViewController and Destinations need modification. This variety of techniques is illustrated next in Figure 83.

One way for the ViewController 8300 to change data 8302 is implicitly through the ApplicationMediator 8304. The ApplicationMediator will have a named method to get the appropriate field, such as CustomerName as shown.

The setData method is called for a ViewEvent 8306 and the ViewEvent is fired. The ApplicationMediator 8304 changes the data through a property change. The other alternative is to change the data explicitly through the
5 ViewController 8300. A put operation on the appropriate field is attached to an instance of the data. This is sent to the data via a ViewEvent. The operation data.put is a property change event.

In general, the developer should keep the data
10 "thin." This means the number of classes to represent data should be kept small and the size should be kept small to facilitate network transfer. For some applications, it is appropriate to use lists for "drill down" or cached data to facilitate disconnected mode.

15

D. TopListener

With reference now to Figure 84, a flowchart of a process used in a TopListener is depicted in accordance with a preferred embodiment of the present invention. The
20 process begins by an ApplicationMediator receiving a ViewEvent from a ViewController (step 8400). A determination is made to whether system service is needed (step 8402). If system service is not needed, the process terminates. Otherwise, a TopEvent is created with the
25 proper major codes (step 8404). Thereafter, the TopEvent is fired (step 8406). After, a set of TopListeners is invoked (step 8408). This set of TopListeners may include a single TopListener or many TopListeners. The TopListener receives the TopEvent (step 8410).
30 Thereafter, a determination is made as to whether the major code of the TopEvent is EXIT (step 8412).

If the major code is exit, then the application is exited (step 8414) with the process terminating thereafter. Otherwise, a determination is made as to whether the major code for the TopEvent is MESSAGE (step 5 8416). If the major code for the TopEvent is message, then the message is displayed for the application (step 8418) with the process terminating thereafter.

If the major code is not a message, then a determination is made as to whether the major code is a 10 TITLE (step 8420). If the major code for the TopEvent is a title, then the title is displayed for the application (step 8422) with the process terminating thereafter. Otherwise, a determination is made as to whether the major code for the TopEvent is to execute (EXEC) (step 15 8424). If the major code is execute, then the data in the TopEvent specifies an application to be launched (step 8426) with the process terminating thereafter. If none of these major codes are present, the process terminates without taking any action.

20

E. PlacementListener

With reference now to Figure 85, a flowchart of a PlacementListener is depicted in accordance with a preferred embodiment of the present invention. The 25 process begins by the ApplicationMediator receiving a ViewEvent from a View Controller (step 8500). Thereafter, a determination is made as to whether ViewController needs to be placed (step 8502). If the ViewController does not need to be placed, the process 30 terminates. Otherwise, a PlacementEvent is created with the appropriate major codes (step 8504). Thereafter, the

ApplicationMediator fires the PlacementEvent (step 8506). A set of PlacementListeners is then invoked (step 8508). In the depicted examples, the set of PlacementListeners may be a set of one PlacementListener or many

5 PlacementListeners.

Thereafter, the PlacementListener receives the PlacementEvent (step 8510). A determination is made as to whether the major code for the PlacementEvent is ADD (step 8512). If the PlacementEvent is add, then the

10 ViewController's component is added to a visual container and ViewController is repainted (step 8514) with the process terminating thereafter. With reference to (step 8512), if the major code is not add, then a determination

15 is made as to whether the placement code is removed in the PlacementEvent (step 8516). If the PlacementEvent is removed, then the ViewController's component is removed from the visual container and repainted (step 8518) with the process terminating thereafter. Otherwise, the process also terminates.

20 With reference now to Figures 86-F3, flowcharts illustrating processes used in a ViewController are depicted in accordance with a preferred embodiment of the present invention. Turning first to Figure 86, a flowchart illustrating handling of AWTEvents by a

25 ViewController is depicted in accordance with a preferred embodiment of the present invention. The process begins by receiving an AWTEvent (step 8600). The Event is handled by manipulating the GUI according to the GUI specifications (step 8602). Then, a determination is

30 made as to whether the AWTEvent.source is a component to validate (step 8603). If the determination is no, then

another determination is made as to whether a ViewEvent is needed (step 8604). If a ViewEvent is needed, then a ViewEvent is created (step 8606). Thereafter, the ViewController is disabled (step 8608) and the ViewEvent
5 is fired (step 8610) with the process then returning.

With reference again to step 8604, if a ViewEvent is not needed, the process also returns.

F. ValidationRule

10 With reference again to step 8602, if the AWTEvent.source is a component to validate, the process then determines whether the validation level is ValidationRule.FOCUS and the AWTEvent is a focus event (step 8612). If the answer to this determination is yes,
15 then the ValidationRules are applied (step 8614). A determination is made as to whether the application was successful (step 8616). If the application was successful, then a determination is made as to whether a validation level is equal to ValidationRule.COMPONENT and
20 whether the AWTEvent is an input event (step 8618). This determination is made directly from step 8612 if the validation level is not set equal to ValidationRule.FOCUS. The AWTEvent is not a focus event. If the determination is that the validation level is a
25 ValidationRule for a component and the AWTEvent is an input event, then the ValidationRules are applied (step 8620) and a determination is made as to whether the rules were successfully applied (step 8622).

If the rules were successfully applied, then a
30 determination is made as to whether the validation level is equal to ValidationRule.VIEWEVENT (step 8624). If

this determination is no, the process then returns to step 8604 as described above. Otherwise a determination is made as to whether a ViewEvent is needed (step 8626). If a ViewEvent is needed, then ValidationRules are applied (step 8628). A determination is made as to whether the ValidationRules were successfully applied (step 8630). If the ValidationRules were successfully applied, the process then proceeds to step 8606 as previously described.

- 10 In any of the steps in which a determination is made as to whether a ValidationRule was successful (steps 8616, 8622, and 8630), if the application of the rules was unsuccessful, then an error message is created (step 8632) with the process then returning. Steps 8612 and
- 15 8618 describe the various times the ViewController will validate user input. If the setValidationLevel of a VC is set to COMPONENT, then validation is performed on every input event for a component (i.e. key strokes, so only valid keys are allowed to be inputted). If the
- 20 validationLevel is FOCUS, then validation is performed when the user hits the Tab key or uses the mouse to move focus out of the current component. If the validationLevel is VIEWEVENT, then validation is performed when a view event needs to be fired. This
- 25 situation typically occurs when the user hits the "Ok" or "Next" button for further processing. If a view event is not needed in step 8626, the process returns.

- With reference again to step 8618, if the determination is such that the validation level is not a
- 30 ValidationRule for a component and the AWTEvent is not an input event, then the process proceeds to step 8624 as described above.

With reference now to **Figure 87**, a flowchart illustrating application of **ValidationRules** is depicted in accordance with a preferred embodiment of the present invention. **Figure 87** is a more detailed description of steps 8614, 8620, and 8628 in **Figure 86**. The process is applied for each rule defined for the component (step 8700). A determination is made as to whether **ValidationRules** are present that have not been applied (step 8702). If more rules are present, then the **ValidationRule(i)** is applied to the component's value (step 8704). A determination is made as to whether a successful application has occurred (step 8706). If a successful application has occurred, the process returns to step 8700 to select the next rule.

With reference again to step 8706, if the application of the rules are unsuccessful, then an exception is thrown (step 8708) with the process then returning. With reference again to step 8702, if a determination that more **ValidationRules** are not present for processing, the process also returns.

G. ViewEvent

With reference now to **Figure 88**, a flowchart of a process for firing a **ViewEvent** is depicted in accordance with a preferred embodiment of the present invention. The process begins by selecting a **ViewListener** (step 8800) that has not yet been processed from the list of **ViewListeners** that have been added to using **addViewListener**. Thereafter, a determination is made as to whether **ViewListeners** are present for processing (step 8802). If **ViewListeners** are present, then invoke the

method `ViewListener(i).viewEventPerformed` (step 8804). This method is implemented by a `ViewListener` (such as an `ApplicationMediator`) to process the `ViewEvent` passed. Typically, this processing of the `ViewEvent` will occur on
5 a separate thread. Also, the `ViewListener` may decide to "consume" the event by calling the `consume()` method on the `ViewEvent`. This essentially specifies that the `ViewEvent` should not be sent to any other `ViewListeners` left to be processed.

- 10 Thereafter, a determination is made as to whether the `ViewEvent` has been consumed (step 8806). If the `ViewEvent` is not consumed, the process returns to step 8800 to select another `ViewListener`. Otherwise, the process returns. With reference again to step 8802, if
15 more `ViewListeners` are not present, the process also returns.

H. Data Refresh

- With reference now to Figures 89-98, flowcharts
20 illustrating processes used by an `ApplicationMediator` are depicted in accordance with a preferred embodiment of the present invention. With reference now to Figure 89, a high level flowchart of the process used by the `ApplicationMediator` is depicted in accordance with a
25 preferred embodiment of the present invention. The process begins by creating `ViewControllers` (step 8900). Thereafter, the `ApplicationMediator` responds to events (step 8902) with the process terminating thereafter.

- With reference now to Figure 90, a flowchart of a
30 process for handling `DataEvents` is depicted in accordance with a preferred embodiment of the present invention. The process begins by the `ApplicationMediator` receiving a

DataEvent (step 9000). In response to receiving the DataEvent, the ApplicationMediator will send a call to refresh data for each ViewController handled by the ApplicationMediator (step 9002) with the process
5 terminating thereafter.

With reference now to Figure 91, a flowchart of a process for adding a listener is depicted in accordance with a preferred embodiment of the present invention. The process begins by the ApplicationMediator receiving a
10 request to add a listener (step 9100). The listener type is then found (step 9102). The listener is then added to a vector (step 9104) with the process terminating thereafter.

With reference now to Figure 92, a flowchart of a
15 process for removing a listener is depicted in accordance with a preferred embodiment of the present invention. The process begins by receiving a request to remove a listener (step 9200). The listener type is then found (step 9202). The listener is then removed from the
20 vector (step 9204) with the process terminating thereafter.

With reference now to Figure 93, a flowchart of a process for managing permissions is depicted in accordance with a preferred embodiment of the present
25 invention. The process begins by receiving a request to gather and set permissions (step 9300). The permissions are then processed (step 9302) with the process terminating thereafter.

With reference now to Figure 94, a flowchart of a
30 process for handling ViewEvents is depicted in accordance with a preferred embodiment of the present invention.

The process begins by receiving a ViewEvent (step 9400). Thereafter, the ViewEvent is processed (step 9402). A determination is made as to whether the major code in the ViewEvent is known (step 9404). If the major code is known, then a determination is made as to whether the minor code is known (step 9406). If the minor code is known, the process then handles the specific request (step 9408) with the process terminating thereafter. A specific request may be, for example, a RequestEvent, a request for fresh data, a TopEvent, a PlacementEvent, or there may not be a specific request.

With reference again to step 9404, if the major code or minor code in step 9406 is unknown, the process defaults (step 9410) and terminates thereafter.

With reference now to Figure 95, a flowchart of a process for refreshing object data in an ApplicationMediator is depicted in accordance with a preferred embodiment of the present invention. The process begins by receiving an event (step 9500) (typically a ViewEvent from a ViewController or other ApplicationMediator). A determination is made as to whether the processing of the event received requires a RequestEvent (step 9502). If a RequestEvent is needed, then a RequestEvent is created (step 9504). Thereafter, the RequestEvent is fired (step 9506). The ApplicationMediator can repeat the processes of creating an event, firing it, processing the results and then go on to the next event to create or end. The process then returns to step 9502.

With reference again to step 9502, if a RequestEvent is not needed for the event, a determination is then made

as to whether a TopEvent is needed (step 9510). If a TopEvent is needed, a TopEvent is created (step 9512) and the TopEvent is fired (step 9514). Thereafter, the process proceeds to step 9508 as described above.

- 5 With reference again to step 9510, if a TopEvent is not needed, a determination is then made as to whether a PlacementEvent is needed (step 9516). If a PlacementEvent is needed, then a PlacementEvent is created (step 9518). Thereafter, a ViewController is
10 selected (step 9520). The PlacementEvent is fired using a component from the ViewController (step 9522) with the process then proceeding to step 9508.

- With reference again to step 9516, if a PlacementEvent is not needed, then a determination is
15 made as to whether a data refresh is needed (step 9524). If a data refresh is needed, then the data is accessed (step 9526). Thereafter, the refresh method on the ViewController or ApplicationMediator is called. (step
20 9528) with the process then proceeding to step 9508 as described above. With reference again to step 9524, if a data refresh is not needed, then the process terminates.

- With reference now to Figures 96 and 97, diagrams illustrating processes used to refresh object data is depicted in accordance with a preferred embodiment of the
25 present invention. The data coming in can be one object or an object containing multiple objects, such as a Vector or more generically, a list. This data is represented by the type "Object data;" the process of data refresh includes seeing if the data is a singleton
30 and if data is handled (a recognized type), then format it (cast) into the recognized type and then call refresh

again in the same view controller. For example, refresh(Object data) yields refresh(Customer data) if this data type is handled (a recognized aggregate type such as Vector), then the data is formatted (cast) into
5 the recognized type. Then refresh is called again in the same view controller. For example, refresh(Object dataAggregate) yields refresh(Vector data).

If the data is an aggregate (not a singleton), and if the data is not handled, then iterations are performed
10 over the aggregate and refresh is called for each object in the aggregate. For example, refresh(Object dataNotSupportedAggregate) yields refresh(Object dataNotSupportedAggregate(i)) for each i.

When a singleton data type is supported, has been
15 cast, and a new refresh has been called, there are a plurality of actions that can result. These actions include, for example: updating one or more components, containers or beans; updating the reference to the local data model; application of ValidationRules again; and
20 even the firing of a new ViewEvent. Thus, for each refresh in the plurality of refreshes possible from the initial refresh, a plurality of actions can result from the plurality of possible actions.

With reference now to Figure 96, a flowchart of a
25 process for refreshing data in an ApplicationMediator is depicted in accordance with the preferred embodiment of the present invention. The process begins by receiving a call to refresh object data (step 9600). Thereafter, an access of data is performed (step 9602). Data is
30 accessed by the ApplicationMediator rarely. The reasons to access data in an ApplicationMediator arise around the need to activate another sub ApplicationMediator or a

ViewController. Thereafter, ApplicationMediator is selected for processing (step 9604). This step is used to identify unprocessed ApplicationMediators for processing. Thereafter, an

- 5 ApplicationMediator(i).refresh(Object data) is performed (step 9606). Thereafter, a determination is made as to whether more ApplicationMediators are present for processing (step 9608). If more ApplicationMediators are present, the process returns to step 9604. Otherwise, a
- 10 ViewController is identified for processing (step 9610). This step is used to select unprocessed ViewControllers for processing. A ViewController (i).refresh(Object data) is performed (step 9612). This step is described in more detail in Figure 97 below. Thereafter, a
- 15 determination is made as to whether more ViewControllers are present for processing (step 9614). If additional ViewControllers are present, the process returns to step 9610. Otherwise, the process returns.

- With reference now to Figure 97, a flowchart of a
- 20 ViewController refresh process is depicted in accordance with the preferred embodiment of the present invention. Figure 97 is a more detailed description of step 9612 in Figure 96. The process begins by receiving a request to refresh (Object data) (step 9700). Thereafter, a
 - 25 determination is made as to whether the data is a singleton (step 9702). If the data is not a singleton, then a determination is made as to whether the data is handled as an aggregate (step 9704). If the data is to be handled as an aggregate, then a determination is made
 - 30 as to whether additional objects are present in the aggregate that are unprocessed (step 9706). If

additional objects are present, then the use object process is performed (step 9708). By using an object, an unprocessed object is selected with a recursive call of refresh on that object. Once that object (and all of its
5 objects) are processed, the next object is then selected. Thereafter, the process returns to step 9706. When no more objects are present, the process returns.

With reference again to step 9704, if the data is not to be handled as an aggregate, the process then
10 proceeds to step 9712 as described below.

With reference again to step 9702, if the data is a singleton, a determination is made as to whether the data is an instance of a recognized type T (step 9710). Type "T" is any Java data model type that the JTC systems is
15 currently supporting. For example, you may support a type Object data; and a type KeyValue data; Until you then add support for XMLData data, T cannot be of that type. Once the code is added for XMLData type, T can be of that type. If the data is an instance of a recognized
20 type T, then the data is cast into a recognized type T (step 9712). Thereafter, a call refresh for type T is made (step 9714). This call may be, for example, refresh (T data). Thereafter, the local state is updated (step 9716), and the GUI is updated (step 9718) with the
25 process then returning. With reference again to step 9710, if the data is not an instance of a recognized type then an ignore or error is generated (step 9720) with the process then returning.

With reference now to Figure 98, a flowchart of a
30 process used to process RequestEvents is depicted in accordance with a preferred embodiment of the present invention. The process begins by the firing of a

RequestEvent (step 9800). Thereafter, the process will select a RequestListener (step 9802). A determination is made as to whether more RequestListeners are present for processing (step 9804). If a RequestListener is present, then the process executes RequestListener(i).requestEventPerformed (step 9806). Thereafter, a determination is made as to whether the RequestEvent has been consumed (step 9808). If the RequestEvent has not been consumed, the process returns to step 9802 to select another RequestListener for processing. Otherwise, the process returns. With reference again to step 9804, if additional RequestListeners are not present for processing, all of the RequestListeners have been processed and the process returns.

I. Hierarchical ApplicationMediators

The present invention provides for the use of hierarchical ApplicationMediators for handling events from ViewControllers. In such a system, ApplicationMediators are arranged in a hierarchical fashion in which different ApplicationMediators may handle different types of events or perform different functions. If graphed, the ApplicationMediators would look like the tree in which nodes represent ApplicationMediators. Events would be received by the lowest level ApplicationMediators, those farthest away from the root. One event will percolate up the graph and then the event might cause actions to a group of ApplicationMediators from the set of ApplicationMediators which in turn may results in a set of actions from the plurality of view controllers. Another way to view it, "event" moves up the graph and then come back down in

numerous possible paths.

Each node in the graph is an ApplicationMediator. View controllers. The partitioning of the nodes are based on some logical heuristic. On example is the
5 higher the node, the higher semantic level the code has.

With reference now to Figure 99, a flowchart of an initialization process for creating hierarchical ApplicationMediators is depicted in accordance with a preferred embodiment of the present invention. The
10 process begins by creating a Top ApplicationMediator (step 9900). TopListeners are added (step 9902), and RequestListeners are added (step 9904). Thereafter, PlacementListeners are added (step 9906) and ViewListeners are added (step 9908).

15 The concept here is that when the application starts the top level AM is provided with the highest level listeners. Then, any Ams created as children AMs are initialized with the same listeners as specified by the top level AM. Once the Listeners are created they will
20 wait to be called back via a fireEvent invocation...topEventPerformed(), viewEventPerformed(), requestEventPerformed(), placementEventPerformed() etc

Next, a second level ApplicationMediator is created (step 9910), and the TopListeners are cloned (step 9912).
25 The RequestListeners are cloned (step 9914), and the PlacementListeners are cloned (step 9916).

Cloning means to create an exact copy, which means that the copy has its own specific reference while a non-cloned copy would have the same reference as the
30 original, which means that if a person has a dollar amount and choose to copy the amount into another object without cloning it then any changes made to the copy will

also affect the original, while the original is cloned rather than just copied, any changes to the cloned object will only affect the clone. Therefore, the reason to clone the listeners here is so that children

5 ApplicationMediators do not have the same Vector instance as the parent ApplicationMediators. If the Vector instance was the same, then the children ApplicationMediators and top level ApplicationMediator would always share the same lists of listeners.

10 Thereafter, the top ApplicationMediator is added as a ValueEventListener to the second level ApplicationMediator (step 9918). A determination is then made as to whether a ViewController is to be added (step 9920). If a ViewController is not to be added, then a determination
15 is made as to whether additional second level ApplicationMediators are to be added (step 9922). If additional second level ApplicationMediators are to be added, the process then returns to step 9910. Otherwise, a determination as to whether a lower level

20 ApplicationMediator is to be added (step 9924).

 If a lower level ApplicationMediator is to be added, then the lower level ApplicationMediator is created (step 9926). The TopListeners are cloned (step 9928) and the RequestListeners are cloned (step 9930). Thereafter, the
25 PlacementListeners are cloned (step 9932). Then, the upper level ApplicationMediator is added as the ValueEventListener (step 9934). The upper level ApplicationMediator is the ApplicationMediator in the level above the one created.

30 If there are no more lower level ApplicationMediators then the initialization process completes and application is ready to begin. Also, lower

level ApplicationMediators and ViewControllers may be created at any time while the application is running.

Thereafter, the process returns to step 9920. With reference to step 9920, if a ViewController is to be added, then the ViewController is created (step 9936). Then, the parent level ApplicationMediator is added as the ViewListener (step 9938) with the process then proceeding to step 9922.

A parent level ApplicationMediator is the application mediator that manages several ApplicationMediators and/or ViewControllers. The usage for this is when functions of an application are nested. For example, assume an application has a function to display an interest rate calculator. This calculator has maybe several screens associated with it. Therefore, an CalculatorApplicationMediator is created to manage those screens. Now, assume that this calculator function will be used by two different higher-level processes such as a Auto Loan calculator and a Home Mortgage calculator. So the AutoLoanAM and HomeMortgageAM would be parent AMs for the CalculatorAM. The parent/child APPLICATIONMEDIATOR communication is done through ViewEvents and refresh just like APPLICATIONMEDIATOR to VIEWCONTROLLER communication. In other words, a child ApplicationMediator represents a child or sub-process that can be re-used in many places.

With reference now to Figure 100, a flowchart of a process for handling events in a hierarchical ApplicationMediator system is depicted in accordance with a preferred embodiment of the present invention. The process begins by receiving user action input (step 10000). A ComponentEvent is created and fired (step 10002). The ViewController receives the ComponentEvent

(step 10004). A determination is then made as to whether the ComponentEvent is a ViewEvent (step 10006). If the ComponentEvent is not a ViewEvent, the process returns to step 10000. Otherwise, the ViewController creates and
5 fires a ViewEvent (step 10008). The ViewController's parent ApplicationMediator receives the ViewEvent (step 10010).

A determination is made as to whether the ApplicationMediator is able to handle the ViewEvent (step
10 10012). If the ApplicationMediator is able to handle the ViewEvent, the ViewEvent is processed (step 10014). Thereafter, a determination is made as to whether the processing has completed (step 10016). If the processing has completed, the process returns to step 10000.
15 Otherwise, the ViewEvent is refired (step 10018). The parent ApplicationMediator to the current ApplicationMediator receives the ViewEvent (step 10020) with the process then returning to step 10012.

With reference again to step 10012, if the
20 ApplicationMediator is not able to handle the ViewEvent, the process proceeds directly to step 10018 as described above.

J. Virtual Application Mediators

With reference now to Figures 101-104, flowcharts
25 illustrating state encoding in an ApplicationMediator is depicted in accordance with a preferred embodiment of the present invention. An ApplicationMediator can be built as a state machine loaded by properties file, which describes other ApplicationMediators. These other
30 ApplicationMediators are also referred to as Virtual ApplicationMediators. The present invention employs an

Docket No. 99-339

encoding language to describe the state transitions of an ApplicationMediator. Encoding is provided for ViewEvents, refresh of data, SettingProperties, SettingPermissions, and for methods that may alter the
5 ApplicationMediator state. The actual Virtual ApplicationMediator used may be described in and passed in SetPropertyies.

An AWTEvent is generalized with increased semantics first as a ViewEvent, then as a RequestEvent,
10 PlacementEvent and TopEvent.

Numerous heuristics may be employed that can determine how to solve mediation, such as how to encode a RequestEvent, why a RequestEvent and not a TopEvent or ViewController action.

15 More than one semantic event can be generated for each ViewEvent.

The Application Mediator interface provides default behavior to manage ViewControllers, ApplicationMediators, and to add/fire/remove PlacementListeners, ViewListeners,
20 and a TopListener. The ApplicationMediatorImpl implements methods in the ApplicationMediator and JTC interfaces.

A dispatching state machine is used to manage all of these actions. As indicated by the name, a state machine
25 consists of a finite set of states, a set of possible transitions between states, and, optionally, one or more actions to be performed when a transition is made between two states. State recognition will be based on the object being monitored. For example, for a ViewEvent the
30 state is typically determined by a subset of the source of the event, the major code, the minor code and possibly the value of the data reference. Examples of possible

actions are to make an existing ViewController visible, to create/fire a PlacementEvent, to create/fire a RequestEvent, or to create/fire a TopEvent. It is possible to "hand build" the finite state machine for every ApplicationMediator, but this route is very tedious.

Instead, according to the present invention, the virtual ApplicationMediator builds the finite state machine from a set of transition rules that recognize the current state and if matched, trigger a state transition and associated actions. In general, it is impractical to build a language within a language. If the encoding of the transition rules included control statements (i.e. if/then/else, while, switch) it is better to implement the ApplicationMediator in the programming language at hand, Java. If the definition of the transition rules are indeed simple, not control structures and the ApplicationMediator has little state except the Vectors of its ApplicationMediators, ViewControllers, Properties, Permissions, and Resources, then encoding becomes simple, flexible, powerful and easy to manage. The resulting benefit is that one set of ApplicationMediator code can be used to implement numerous virtual ApplicationMediators simply by changing the transition rules via the setProperties method.

In Figure 101, a flowchart for a process for building a state machine is depicted in accordance with a preferred embodiment of the present invention. The process begins by loading a configuration file of ApplicationMediators state stanzas (step 10100). A multi-dimensional list of the configuration file is built (step 10102). A multi-dimensional list has more than one

dimension, traditional list are a single dimension implementation such as a list of numbers, or collection of elements. Multi-dimensional list would have, as its elements, containers that also represent containers. For example, a list of lists. This is clearly a list, yet the elements themselves are lists. This means a particular list in one dimension may be referenced while accessing the elements of that list in another dimension. Example colorList is a list of colors. To reference a color, a call would be made, such as colorList[2] and the color that is stored at location 2 would be obtained. If it is desired to store several list of list of colors and access to a particliar color that is in color list 2, at location 4 is desired, the color would be referenced as listOfColors[2][4]. Thereafter, the state machine processes events and calls (step 10104) with the process terminating thereafter.

Turning now to Figure 102, example table entries from the loading of a configuration file for a virtual ApplicationMediator is depicted in accordance with a preferred embodiment of the present invention.

With reference now to Figures 103 and 104, an access state machine used to determine whether processing of a ViewEvent is needed is depicted in accordance with a preferred embodiment of the present invention. The process begins by receiving a ViewEvent (step 10300). After the ViewEvent.source is located (step 10302), a determination is then made as to whether the source is in a virtual ApplicationMediator list. (step 10304). If the ViewController is not located in the dispatch table, the process returns. Otherwise, a determination is made as to whether a minor code has been set for the ViewEvent

(step 10306). If a minor code has been set for the ViewEvent, then the table entries locate source /ViewEvent.major/ViewEvent.minor tuple (step 10308).

The process then determines whether more table
5 entries are present for processing (step 10310). If more table entries are not present for processing, the process returns. Otherwise, table entry (i) is used (step 10312). A determination is made as to whether table entry(i) has a major equal to ViewEvent.major and/or
10 entry(i).minor is equal to ViewEvent.minor (step 10314). If the table does not have a major equal to the major for the ViewEvent, the process returns to step 10310 to see if more table entries are present for processing. Otherwise, a determination is made as to whether more
15 action entries are present for entry(i) (step 10316). If additional action entries are not present, the process also returns to step 10310.

With reference again to step 10306, if the ViewEvent is not a minor code set, then the table entries matching
20 source and the major pair for the ViewEvent are located (step 10318) with the process then returning to step 10310.

With reference again to step 10316, if more action entries are present for entry(i), the process then
25 proceeds to **Figure 104** and determines whether action(i) is equal to RE (step 10320). If action(i) equals RE, then a RequestEvent is created with action(i).major, action(i).minor, ViewEvent.data (step 10322). Thereafter, the RequestEvent is fired (step 10324) with
30 the process then returning. With reference again to step 10320, if action(i) does not equal to RE, then a

determination is made as to whether action(i) equals TE (step 10326). If action(i) equals TE, then a TopEvent is created with an action(i) major and an action(i) minor (step 10328). Thereafter, the TopEvent is fired (step 5 10330) with the process then returning.

With reference again to step 10326, if action(i) does not equal TE, then a determination is made as to whether action(i) is equal to PE (step 10332). If action(i) is equal to PE, then a PlacementEvent is 10 created with an action(i) major and an action(i) minor along with a ViewEvent source (step 10334). Thereafter, the PlacementEvent is fired (step 10336) with the process returning. With reference again to step 10332, if action(i) does not equal to PE, then determination is 15 made as to whether action(i) is equal to VIEWCONTROLLER (step 10338). If action(i) is equal to VIEWCONTROLLER, then a refresh data is performed on the source for the ViewEvent (step 10340) with process then returning. With reference again to step 10338, if action(i) does not 20 equal ViewController, then an error has occurred and an error is generated (step 10342) with the process then returning

K. Serialization/Deserialization

When communications take place between the client and 25 the server, it is convenient to "pack" an object into a single entity, send it over the network, and then "unpack" it at the other end. The Serializer packs the object for transmission and the Deserializer unpacks it at the other end.

30 With reference now to Figure 105, a diagram of a serializer system is depicted in accordance with a

preferred embodiment of the present invention.

Serializer system 10500 is used to serialize data for transfer. With reference to Figure 106, a deserialier is depicted in accordance with a preferred embodiment of the present invention.

When an object is serialized, the object is being written to an external source. The size of the data array is first written and then the data is sent to a serializer for serialization. Typically, a serializer will look up the code for the data element, write the code to the stream, and then look up the serializer for the element. If the serializer for the element exists, then the serializer is allowed to write the element. Otherwise, if the element is "externalizable", it is allowed to write itself. In this last case, it is a call to the method writeExternal on the data element. If the element does not have a serializer and is not externalizable, then it is written as a standard object. Such a writing of the element as a standard object is expensive in terms of serialization space and time and should not happen under normal circumstance.

When an object is read, the size of the data array is read. Thereafter, a base serializer class is used to read each data element. The code for the data element is read and looked up to see if a serializer exists for the code. If a serializer exists, then the serializer reads the data elements and creates the object. Otherwise, an instance of the element type is created. If the element is externalizable, the element is allowed to read itself. This may be accomplished through a call to readExternal on the data element. Otherwise, the data element is read into an object.

The default serialization provided by Java writes the class name of the data element then the attribute name and the value for each attribute of the data element. The size benefit provided by the present invention comes from writing a code representing the element type instead of the type name and the attributes that are required to recreate the data objects without writing the names of the attributes. For example, an integer is written as follows: The class name [java.lang.Integer], the attribute name [value java.lang.Number], the attribute value [00 00 00 08] size: 81 bytes. Alternatively enhanced serialization: the code [01], the value [00 00 00 08] size: 11 bytes.

With reference now to Figure 107, a diagram illustrating an object array is depicted in accordance with a preferred embodiment of the present invention. Typically user defined data classes extend a class similar to that shown in Figure 107. Specifically, Figure 107 illustrates an object array that contains all the attributes of the user-defined data. The subclasses just provide getter/setter methods (i.e. getXXX() and setXXX() methods where XXX is a particular attribute name). These getter and setter methods just access the object array in Figure 107 using getData and setData methods based on an index for the attribute. This is a common extension of using an indexed-array data model combined with an attribute-based object model. The advantage of this model is to reduce the serialization size by removing the attribute names that are typically outputted in the default Java serialization mechanism.

Turning now to Figure 108, a diagram illustrating code used in a serialization method is depicted in

accordance with a preferred embodiment of the present invention. The mechanism of the present invention adds to the serialization method 10800 provided in Figure 108.

To use the special serialization, any and all

5 user-defined data classes can extend the BaseDataS class 10800 illustrated in Figure 108. In Java, the default Java serialization for an object can be overridden by implementing the Externalizable interface illustrated in Figure 109 and implementing the methods readExternal and
10 writeExternal for reading/writing from/to input/output stream.

Method 10800 uses the BaseSerializer class, as illustrated in Figure 110 for implementing default serialization for many of Java's base classes and even
15 some of Java's collection classes. Also, the user can specify a serializer for other data classes by implementing the SerializerIF interface 10900 in Figure 109 and listing the serializer's class name on the same line after the data class name in the ClassNames.ini file
20 separated by a space. If no serializer is specified and none are provided by the BaseSerializer 11000, then the default Java serialization method for that class is used.

Using this new serialization mechanism coded in the read/write External methods of BaseDataS method in Figure
25 108 and BaseSerializer method 10800 in Figure 109 provides the benefits of reduced serialization size. The combination of the two mechanisms above provide the 80% reduction in serialization size.

In these examples, the serialization process of the
30 present invention is able to reduce the serialization size further by completely removing any full-path class name strings that are typically outputted in the default

Java serialization. Of course, depending on the implementation, other information may be removed and processed a similar manner to the full-path class name strings.

- 5 The number and length of these class name strings can incur a high amount of overhead for serialization. The only disadvantage is that the actual class names must be provided on both sides of serialization (reader/writer). The class names are hashed and the hash
- 10 code is stored in the serialization instead of the actual full class name. (This is why the classes need to be listed in the ClassNames.ini file and provided on either side of the client/server that needs
- 15 serialize/unserialize the data). Another way to reduce this dependency is to automatically send the class names once before any other communication between client/server is done.

- 20 The ClassNames.ini file contains a sequence of lines that are either a class name for a user-defined data class or a class name followed by a serializer's class name if the user created their own serializer for that particular data class. If no serializer is provided and the data class is not in the BaseSerializer, then the default Java serializer for the data class is used. The process of
- 25 reading this file produces a table that contains the class name, a hash code, and a serializer. The flowchart in Figure 111 shows how the ClassNames.ini file is read.

- 30 Since the read will be buffered, an empty BufferedRead object is declared (step 11102). A FileReader object is created and passed to the constructor for the BufferedRead object (step 11104). Each line of the file is read sequentially (step 11106). A

determination is then made as to whether there is additional data in the file (step 11108). If so, control passes to the finalization code (step 11130). Otherwise, the line is trimmed (step 11110), which means all leading and trailing spaces are removed. Next, a determination is made as to whether the line length is 0 or it begins with a # (indicating a comment) (step 11112). If not, the process returns to step 11106 where another line is read. For lines with data, a string tokenizer is created and the first token is read (this is the class name) and the Java method hashCode() is applied to the class name to find an associated integer value (step 11114); this integer value is used for hash table processing. StringTokenizer is a built-in Java class that breaks a string into its components, much like a sentence could be broken into separate words. Next, an empty serializer is created (step 11116).

A determination is then made as to whether there is another token (step 11118). If so, then it is fetched (this is a user-defined serializer name) (step 11120) and a serializer is created using this name (step 11122). If this causes an exception, it is ignored (step 11124) and the class name, hash code, and serializer name are added to the table (step 11126). Control returns to step 11106 to read the next line from the file. If an exception occurs at any time while processing the ini file, it is ignored (step 11128). The finalization code closes the file (step 11130); any exception is ignored (step 11132). Then the table is written (step 11134) completing the process of reading the ini file. With reference again to step 11122, if there are no additional tokens in the

tokenizer, the process proceeds directly to step 11126 as described above.

L. Permissions

A permission is a capability that a user is allowed to perform. For example, given a user (defined with a user id, password, location, etc.), this user's capabilities may include running ApplicationMediator 1 and 2, but not 3. Similarly, the same user may be allowed to operate ViewController 1 and 2 in ApplicationMediator 1, but not ViewController 3. Similarly, the same user may be allowed to use all of the GUI components in ViewController 1 in ApplicationMediator 1, except the "next" button, which is disabled.

A permission is the encoding of the above functionality. It is defined as part of the enterprise deployment business processes and security model. Enterprise support to maintain storage of permission data is required for JTC permissions to operate.

For example, using the above description, [Joshua: "AM1=yes AM2=yes AM3=no"] would be defined before deployment and stored in a database and later to be sent to the client. The client program knows that, while the current user is Joshua, AM3 cannot be started.

For another example, [Jacob: "AM1: VC1=yes VC2=yes VC3=no"] would be defined before deployment and stored in a database and later to be sent to the client. Now the client program knows that, while the current user is Jacob and running AM1, Jacob cannot use VC3.

For example, [John: "AM1.VC1: customerlist=yes ok=yes previous=yes namefiled=yes next=no"] would be defined before deployment and stored in a database and to be sent to the client. Now the client program knows

that, while the current user is John and running AM1, and in VC1, the customerlist, ok button, previous button, and namefield are enabled while the next button is disabled. The JTC ApplicationMediators and ViewControllers alter
5 their internal state when setPermissions method is called. They do not hard code user permission information. They do check a system specific user session and compare it to their permission state to determine what actions to alter for ViewEvents, PlacementEvents,
10 etc.

Likewise, possibly at build time, the ApplicationMediators and ViewControllers, when called with getPermissions, will return the sets of "keys" that they will react to at runtime. The actual value of the
15 keys (for example, "yes" above) is implementation and business specific.

With respect to permissions, user will typically login to an Application. Login validation is typically provided by external services. The user may have a role
20 based on various attributes, such as user ID, password, location, or time. After logging in, the user will execute the Application as defined by the role. Sometimes, functions provided by another role not accessible by the current user may be required. For
25 example, a cashier may need to void a transaction, requiring the manager to issue an override. For another example, a salesperson may need a branch banking loan officer to approve a loan. In such a case, a new user will login to supplement or replace the current user.
30 The new user can execute the Application according to the new role. In such a case, more functionality may be enabled. Thereafter, the new user will logout. The

architectural pattern of the present invention provides a key/value pair of permissions model to support such roles. In such a case, ApplicationMediators may enable and disable ViewControllers while ViewControllers enable and disable GUI parts. In such a case, an ApplicationMediator will return a list of "keys" that represent alterable items. These are usually ViewControllers. New ViewControllers will return a list of "keys" that represent alterable items. These are usually GUI related components, containers and beans. A user management system may create a GUI tool to map role definitions to the ApplicationMediator and ViewController permission keys (i.e. IBM's On Demand Server). In such a case, for example, for each key, a well defined value is stored that defines the alterable item. These keys/value mapping are stored in a database. In setting permissions, if a login is successful, the login and service provider will provide the key/value mappings. For example, ApplicationMediators will be passed the key/value mappings. In turn, the ApplicationMediators will pass them to the ViewControllers associated with the ApplicationMediators.

The permission information is almost always retrieved or changed via a change in user login (security). A JTC application may choose to create a simple list of the user login sessions. Each session may contain permissions for the user as well as properties, references to ApplicationMediators and references to ViewControllers. A policy of a single current user, two current users or multiple current users may be implemented. In all cases, the ApplicationMediators and ViewControllers do not know why properties are being set,

they are just set because top level ApplicationMediator receives the properties data from the server or the login session list and calls the setPermissions method.

- Figures 112-119 are a series of flowcharts
- 5 describing the process for getting and setting permissions are depicted in accordance with a preferred embodiment of the present invention. With reference now to Figure 112, a high level flowchart of a process for obtaining permissions is depicted in accordance with a
- 10 preferred embodiment of the present invention. The process begins by determining whether a user profile has been created (step 11200). If a user profile has not been created, the process creates a user profile (step 11202). Thereafter, the process returns to step 11200.
- 15 If a user profile has been created, then program permissions are obtained (step 11204). These program permissions are stored as permission keys in a database (step 11206) with the process terminating thereafter.

- With reference now to Figure 113, a flowchart of a
- 20 process for getting program permissions is depicted in accordance with a preferred embodiment of the present invention. The process is iterated for each ApplicationMediator created (step 11300). A determination is made as to whether more
- 25 ApplicationMediators are present (step 11302). If more ApplicationMediators are present that have not been processed, then the permission keys for the particular ApplicationMediator are added to the list (step 11304) with the process then returning to step 11302. With
- 30 reference again to step 11302, if more ApplicationMediators are not present for processing, then a list of keys are returned (step 11306) with the process

terminating thereafter.

With reference now to Figure 114, a flowchart of a process for obtaining ApplicationMediator permissions is depicted in accordance with a preferred embodiment of the present invention. The process begins by iterating for each ViewController created (step 11400). A determination is then made as to whether more ViewControllers are present for processing (step 11402). If more ViewControllers are present, then the permission keys for the ViewController are obtained and added to the list (step 11404) with the process then returning to step 11400.

With reference to step 11402, if more ViewControllers are not present for processing, then the names of the ViewControllers are added to the list formed for each ViewController (step 11406). Next, a list of keys are returned (step 11408) with the process terminating thereafter.

With reference now to Figure 115, a flowchart of a process for obtaining ViewController permissions is depicted in accordance with a preferred embodiment of the present invention. The process begins by iterating for each component, container or bean created (step 11500). A determination is made as to whether more components are present for processing (step 11502). If more components are present, a determination is then made as to whether the component is alterable at run time based on permissions (step 11504). If the component is not alterable at run time, then the process returns to step 11502 to process another component. Otherwise, a key representing the component is created (step 11506). The

key is then added to the list (step 11508) with the process then returning to step 11500. With reference to step 11502, if more components are not present, then a list of keys are returned to step 11510 with the process
5 terminating thereafter.

With reference now to Figure 116, a flowchart of a process for setting permissions is depicted in accordance with a preferred embodiment of the present invention. The process begins by determining whether the user is
10 logged in (step 11600). If the user is not logged in, then a login process is performed (step 11602) with the process then returning to step 11600.

When the user is logged in, program permissions are obtained from the database (step 11604). Thereafter, the
15 program permissions are set (step 11606) with the process then terminating.

With reference now to Figure 117, a flowchart of a process for setting program permissions is depicted in accordance with a preferred embodiment of the present
20 invention. The process begins with iterating through each ApplicationMediator present and selects an ApplicationMediator for processing (step 11700). A determination is then made as to whether more ApplicationMediators are present (step 11702). If more
25 ApplicationMediators are not present, the process then returns. Otherwise, permission keys are set for the particular ApplicationMediator being processed (step 11704) with the process then returning to step 11700.

With reference now to Figure 118, a flowchart of a
30 process for setting ApplicationMediator permissions is depicted in accordance with a preferred embodiment of the

present invention. The process begins by selecting a ViewController for processing and will iterate through each ViewController present (step 11800). Thereafter, a determination is made as to whether more ViewControllers are present (step 11802). If more ViewControllers are present for processing, the process then sets permission keys for the particular ViewController (step 11804) with the process then returning to step 11800. With reference back to step 11802, when additional View Controllers are not present for processing, then for each ApplicationMediator permission key, the value is remembered and applied to the ViewController at one time (step 11806) with the process returning thereafter.

With reference to Figure 119, a flowchart of a process for setting ViewController permissions is depicted in accordance with a preferred embodiment of the present invention. The process begins by selecting a permission key for processing and will iterate for each permission key (step 11900). A determination is then made as to whether more permission keys are present (step 11902). If more permission keys are present, the process then gets the value for the key (step 11904). Thereafter, a value is applied to the particular components (step 11906). The value may be, for example, related to the method setVisible, setEnabled, or setAttribute. The process then returns to step 11900. With reference to step 11902, if more keys are not present, the process then returns.

VIII. Example Patterns

Figures 120-123 illustrate example patterns using

the architectural pattern of the present invention. Turning now to **Figure 120**, a sample pattern for a Uniform resource locator (URL) mapper 12000 is depicted in accordance with a preferred embodiment of the present invention. URL mapper 12000 is a pattern which provides a mapping to URL links that are cached. The user interface 12002 is part of the ViewController. User interactions that are AWT events are translated to ViewEvents that are processed by the ApplicationMediator 12004. These ViewEvents become RequestEvents that are sent via the transporter 12006 to the destination 12008. The RequestEvent responses carry the hypertext markup language (HTML) and are cached.

With reference now to **Figure 121**, a data model using the architectural pattern of the present invention is depicted in accordance with a preferred embodiment of the present invention. In this example, pattern 12100 in **Figure 121** illustrates the use of data models. The user interface 12102 is implemented as multiple classes that can interact with an XML database 12104. ViewEvents generated as a result of user interaction are translated into RequestEvents by the ApplicationMediator 12106. These RequestEvents are sent to the destination 12110 via the transporter 12108. The RequestEvent responses carry XML objects that are put into the XML database.

Turning next to **Figure 122**, a diagram illustrating interaction with logged data objects processed in a streaming environment is depicted in accordance with a preferred embodiment of the present invention. In this example, the pattern 12200 in **Figure 122** illustrates interaction with live data objects processed in a

streaming environment. The user interface 12202 is implemented as multiple classes that can interact with a Customer database 12204. There is also asynchronous interaction with CurrencyCounter 12206 that keeps track of the data objects. ViewEvents generated as a result of user interaction are translated into RequestEvents by the ApplicationMediator 12208. These RequestEvents are sent to the destination 12212 via the transported 12210. This is an Enterprise Java Beans application where EJB objects are sent back as part of the responses. These objects affect both the Customer database 12204 and the CurrencyCounter 12206. The Customer data 12204 can also be accessed remotely through an Remote Method Invocation (RMI) server 12214. The CurrencyCounter 12206 also communicates asynchronously with an RS232 port 12216.

A JTC program and its elements (i.e. Transporters, Destinations, ApplicationMediators, ViewControllers, components/containers, beans and the main program) all implement the JTC interface. If a reference to the main program is available, a call may be made to getJTCs and a list of all the objects that main allocates that implement the JTC interface are returned. This mechanism is independent of the AWTEvent, ViewEvent, RequestEvent paths.

Iteration can be performed over this list and a determination can be made what the object types with the type of the object, the type of listeners it supported by the objects can be identified. Therefore, a process can be added to objects as a listener as long as the callback method is supported. This can be done for any object type. This normally includes JTC objects and AWT (includes AWT and JFC) objects.

For any object, numerous events may be generated. These events are associated with an interface. If an object is added as a listener to the object for these events, the object must implement the numerous methods
5 associated with the numerous interfaces to support the numerous events from the various types of objects.

After the event arrives, numerous actions may result ranging from logging, to tracing, to connecting to a network and sending them out of the box, to generating
10 further events. A tracing program can in fact be a JTC program and it can be hooked (traced) while it is tracing another JTC program. A graph-tree of programs may be built where the root is the top level program and the root listens to its children and so forth, until the
15 leaves are reached and they are just running JTC programs.

Although events such as AWTEvent, ViewEvent, RequestEvent, and TopEvent, still occur anywhere in the graph, they do not cause the hooking via getJTCs/getAWTs
20 to start. This is done primarily when the program is launched. It could be initiated in the event path. Alternatively, the TopEvent may cause the hooking to start. For example, the user hits "help" in a ViewController that causes a ViewEvent to a RequestEvent
25 to launch a request for help and it also sends a TopEvent to trace the program and send the events to a call center over the network.

With reference next to Figure 123, a diagram illustrating nonintrusive caching, tracing, or logging
30 using an architectural pattern is depicted in accordance with a preferred embodiment of the present invention.

Pattern 12300 in Figure 123 shows how non-intrusive

1 caching, tracing, or logging can be added to any thin
2 client pattern. The main sequence of events from the
3 application perspective is ViewEvents generated in the
4 ViewControllers 12302 being handled by the

- 5 ApplicationMediator 12304 and translated into appropriate
6 RequestEvents. These RequestEvents are passed on to the
7 destination 12308 via the transported 12306. A
8 SnooperApplicationMediator 12310 runs in parallel with
9 the application. The SnoopListener monitors ViewEvents
10 and RequestEvents and, as appropriate, issues
11 RequestEvents to the logging system. The
12 SnoopDestination 12312 can log, trace or cache
13 information on an appropriate storage device 12314.

- 14 Thus, the present invention provides an
15 architectural pattern that may be used to create Internet
16 friendly Java applications, which are small, fast, and
17 flexible. Additionally, through the architectural
18 pattern of the present invention, the object reuse on a
19 client may be between 50 to 100 percent. Additionally,
20 the architectural pattern and methodology of the present
21 invention reduces the time needed to develop an
22 application. By the separation of the functions and
23 processes as described above, a client application may be
24 designed, implemented, and tested as parallel tasks in
25 the client application. By the separation of the
26 functions and processes as described above, a client
27 application may be designed, implemented, and tested in
28 parallel with the server application. Through the
29 present invention, lock-step designing and testing may be
30 eliminated. The addition of new network protocols, new
31 servers and new data models are straight forward and
32 easily performed. Also, changing the graphical user

interface to change the look and feel of an application also is easily performed through the compartmentalization and segregation of duties and functions set out in the architectural pattern of the present invention.

5 It is important to note that while the present invention has been described in the context of a fully functioning data processing system, those of ordinary skill in the art will appreciate that the processes of the present invention are capable of being distributed in
10 the form of a computer readable medium of instructions and a variety of forms and that the present invention applies equally regardless of the particular type of signal bearing media actually used to carry out the distribution. Examples of computer readable media
15 include recordable-type media such a floppy disc, a hard disk drive, a RAM, and CD-ROMs and transmission-type media such as digital and analog communications links.

 The description of the present invention has been presented for purposes of illustration and description,
20 but is not intended to be exhaustive or limited to the invention in the form disclosed. Many modifications and variations will be apparent to those of ordinary skill in the art. For example, although the depicted architectural pattern is illustrated in a Java programming environment,
25 the architectural pattern of the present invention may be applied to other types of programming environments. For example, VisualBasic, C++ and Smalltalk are other programming environments in which the processes of the present invention may be applied. In addition, the
30 description of the classes along with the variables, constructors, and methods are provided for purposes of illustration only. Classes, variables, constructors, and

Docket No. 9-99-339

methods may vary depending on the particular
implementation. Illustration of these classes,
variables, constructors are for illustrative purposes and
not intended to limit the architectural pattern of the
5 present invention. The embodiment was chosen and
described in order to best explain the principles of the
invention, the practical application, and to enable
others of ordinary skill in the art to understand the
invention for various embodiments with various
10 modifications as are suited to the particular use
contemplated.